# FLEXIBLE AND EFFICIENT ACCELERATOR ARCHITECTURE FOR RUNTIME MONITORING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Danny Yu Deng

May 2016

FLEXIBLE AND EFFICIENT ACCELERATOR ARCHITECTURE FOR RUNTIME MONITORING

Danny Yu Deng, Ph.D.

Cornell University 2016

Secure and reliable computing remains an open problem. Just in the past year, security vulnerabilities continued to be discovered in widely-used applications, and remote attacks that take advantage of these vulnerabilities can cause an enormous amount of damage. Runtime monitoring is a promising approach for addressing a wide range of security and reliability problems in a general and transparent fashion. However, software-only approaches have high performance overheads on applications, and using custom hardware to mitigate these overheads can result in rigid schemes that are unable to adapt to new threats. In this thesis, we will describe steps to build an architectural framework for runtime monitoring that can address the drawbacks of previously proposed runtime monitoring schemes by utilizing reconfigurable hardware and optimizing for performance.

## BIOGRAPHICAL SKETCH

Daniel is from Changsha, China and grew up in Mendon, Massachusetts. He received his Bachelor of Science degree in Electrical and Computer Systems Engineering from Rensselaer Polytechnic University in 2002. Prior to Cornell University, he worked for IBM in Burlington, Vermont.

This document is dedicated to my parents and my greatest role models, Kuan Deng and Lihua Zeng.

## ACKNOWLEDGEMENTS

I would like to acknowledge and thank all student and faculty members of CSL for their help through my journey as a graduate student. This journey has been a tremendous challenge, and it has shaped my thinking and my work processes for the better.

I would like to express the deepest appreciation and gratitude to my advisor, Professor G. Edward Suh for giving me the opportunity to work under his guidance. As his first student, Ed gave me the opportunity to work on some really interesting projects. At the same time, Ed also challenged me to become more articulate, to write more coherently, and to gain a more in-depth understanding of the trade-offs in research and design.

I would like to give special thanks to both of my special committee members, Professor José Martínez and Professor Rajit Manohar for their guidance and support. Through their classes and feedback on my work, I gained a greater understanding and appreciation for the nuances of architecture and logic design.

I also thank and acknowledge the members of my research group, the Trusted Systems Group, which I have had the pleasure of working and collaborating with over the time that I spent at Cornell. I thank in no particular order, Andrew Chan, Shahriyar Amini, Raymond Huang, Daniel Lo, Skyler Schneider, Greg Malysa, KK Yu, Tao Nguyen, Luke Ackerman, Mohamed Ismail, Dan Ji, Marc Vac, Teja Panchagnula, and Yao Wang for all of the discussions, insight, and assistance that they have helped me with.

I would like to acknowledge the National Science Foundation, the Air Force, the Office of Naval Research, the Army Research Office, and Cornell University Electrical and Computer Engineering department, for the funding that helped

to make my research possible; and Intel Corporation for the donated equipment that I worked on.

Last but certainly not least, I would to like to thank my parents, Kuan Deng and Lihua Zeng, and my sister, Julia Deng. I would not be able to get to where I am today without their love and support.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

This dissertation presents a co-processing architecture for security and reliability. The architecture combines a traditional processing core with an on-chip programmable fabric. Using an interface that can expose events on the main processing core, the programmable fabric can be dynamically configured to perform monitoring operations on the main processing core. In this work, we will show how such an architecture can perform runtime monitoring in an flexible manner and with low performance overheads on monitored applications.

This research was motivated by the fact that individuals, government, and corporations are increasingly more reliant on using computing technology [164] to manage their assets, critical operations, and connectivity to others. This reliance leads to untold amounts of damage being done when an a system malfunctions [164] or when an exploit is manifested [147]. Despite the efforts of large corporations and research teams to attempt to address these exploits, software vulnerabilities and attacks that take advantage of them are still prevalent [143, 142, 8].

Prior studies in runtime monitoring [110, 20, 51, 88, 101, 121] have shown that it can defend against a broad range of errors. Runtime monitoring can detect errors in the monitored application by tracking the internal state of individual application as it executes and ensuring that certain invariants are always met. In addition, fine-grained or instruction-grained runtime monitoring can be very effective as it has several unique advantages. By monitoring at an instruction granularity [101, 88, 113, 121], fine-grained runtime monitoring has access to highly detailed information regarding dynamic events in the application such

as memory references, address computation, control transfers, and data movement. This advantage allows runtime monitoring to be effectively used to ensure a wide range of security and reliability properties of the system. All told, runtime monitoring can enable sophisticated attack detection and prevention schemes such as fine-grained memory protection [94, 42, 37, 44, 161, 149], array bound checking [44, 33], software debugging support [166], managed language support such as garbage collection [76], hardware error detection [90], and many others. Another advantage of fine-grained runtime monitoring is that, compared to more coarse-grained monitoring [39] and analysis approaches, software errors can be captured earlier and more accurately by fine-grained runtime monitoring approaches.

As an example, researchers identified input channels such as the network an important avenues for an attacker to enter a vulnerable system. Information flow tracking [131, 102, 26, 43] was proposed as a way to defend against control hijacking attacks that originate from the network by tainting data that is affected by network inputs and tracking the use of tainted data. Information flow tracking functions at the granularity of individual instructions to update bookkeeping (set metadata of destination operand based on source operands and the executed instruction) and perform checks. Using metadata in the range of one to several bits for each byte, information flow tracking approaches were shown to be effective in defending against buffer overflow [106], format string [86], cross-site scripting, SQL injection [132, 133], and double free errors.

Fine-grained runtime monitoring is not a new concept, there have been a variety of previously proposed approaches for runtime monitoring. They can be broadly categorized into software, multicore software, and custom hard-

ware approaches. These approaches show tension between performance and the flexibility to adapt to monitoring needs. Runtime monitoring approaches that were built as software platforms [64, 11, 87, 47, 88, 101, 102, 141] can monitor each instruction of the monitored application by adding several instructions for bookkeeping and checking. These approaches were flexible as they leveraged the inherent programmability of general purpose processing cores. However, these approaches presented very high (40-100X) performance overheads on the monitored application [102]. Code optimizations that reduced the flexibility of these approaches were able to reduce the overheads to a lower but still high 5-12X [113, 96]. Such high overheads remained a problem for runtime monitoring as they can hinder the usability of monitored applications.

Multicore and custom-hardware-based approaches were proposed as ways to sustain the functionality of runtime monitoring while reducing their overheads on performance. Multicore approaches functioned by communicating completed instructions from the core running the monitored application to another processing core on a multiprocessing system [108, 125, 28, 162]. Multicore monitoring approaches reduced the performance overheads of runtime monitoring but are inefficient as they incur the area and power overheads of another dedicated processing core. These overheads can be reduced through the use of specialized hardware accelerators that can perform the bookkeeping and checking operations more efficiently [28, 146, 145]. Custom-hardware-based runtime monitoring approaches can have the lowest performance overheads and are the most efficient from an area and energy perspective. However, the adoption of custom hardware can be an expensive proposition. Once fabricated, new usage scenarios and zero-day exploits can quickly render algorithms that are implemented in custom hardware obsolete. These high development costs and the

lack of flexibility makes the use of custom hardware rather impractical.

At a high level, it can be observed that there is a tradeoff between efficiency and flexibility for runtime monitoring approaches. Custom-hardware-based runtime monitoring approaches are the most efficient but have the least flexibility. Software-instrumentation-based approaches have the most flexibility but are the least efficient with high overheads on the monitored application. The key to long-term impact for runtime monitoring is both high efficiency and high flexibility for two reasons. The first reason is that monitoring functions that are too slow will make the application unusable. The second reason is that the set of important monitoring functions will likely change over time as security and reliability requires change for new applications and usage scenarios. In addition, flexibility can be paramount as zero-day attacks target newly discovered vulnerabilities; while protection schemes must change so as to protect against these new threats.

## 1.1 Thesis Statement and Dissertation Roadmap

*This dissertation investigates a co-processing platform that leverages reconfigurable hardware for security and reliability monitoring, that can be a proxy for the flexibility of software-based monitoring approaches while being able to offer the efficiency of custom hardware.*

This research will illustrate the overall co-processing architecture in the context of a single-core processor. We envision that the programmable runtime monitoring framework can be extended to multi-core systems where each processing core of the system would have its own reconfigurable accelerator for

mapping runtime monitoring functions. The rest of the chapter will give an overview of the remaining chapters in this thesis.

## 1.2 Fully programmable hardware accelerator for runtime monitoring

First, we will first present an initial implementation of the co-processing architecture named FlexCore that combines a traditional processing core with an FPGA-like, bit-level reconfigurable fabric. FlexCore can be dynamically reconfigured to perform any type of monitoring or computation that can fit within the fabric. The use of reconfigurable hardware in this fashion deviates from traditional approaches that used reconfigurable hardware to accelerate certain types of computation [150, 65]. The types of accelerators that can be supported by FlexCore are control-coupled to the main computation, can independently access shared memory, and interrupt the main processing core when monitoring requirements are not met. One of the main benefits of this architecture are that it allows hardware monitoring functions to be dynamically added to the system long after the chip has been fabricated.

Our preliminary evaluation with FlexCore showed that the platform was more efficient for runtime monitoring than another dedicated processing core; and had lower performance overheads than pre-existing software-based approaches. The reconfigurable fabric was also quite flexible in being capable of implementing several different types of monitoring approaches that fit within the number of logic elements available on the fabric. To demonstrate that the platform was effective in detecting security attacks, a prototype was built that

runs on a Xilinx FPGA board.

However, the relatively low throughput of reconfigurable fabric used in FlexCore meant that the platform is unable to keep pace with modern, high performance processing cores. In our analysis, we identified two high-level sources of performance overheads: the difference in instruction throughput between the main processing core and accelerators on the reconfigurable fabric; and contention between the main processing core and the accelerator for main memory bandwidth. In the subsequent chapters, we describe techniques that aim to address these sources of overhead.

## 1.3 Improved hardware fabric for runtime monitoring

For the throughput of the reconfigurable fabric, there appears to be a fundamental trade-off between efficiency and programmability. Programmability requires additional hardware so as to provide for choice, but the extra hardware results in longer delays and lower throughput. In Chapter 4, we introduce an optimized co-processing architecture named Harmoni. Harmoni aims to improve performance by carefully restricting the programmability of the reconfigurable fabric such that it can still meet the demands of common monitoring approaches.

In particular, we found that many of the instruction-grained runtime monitoring techniques are built on the notion of tagging, where metadata is bound to program state, and monitoring functionality is tied to the management and checking of metadata. Harmoni shares FlexCore's ability to support monitoring functions that are control-coupled to the main computation. However, by

focusing specifically on monitoring approaches that make use of the tagging, Harmoni is able to achieve higher operating clock frequencies of 1.25GHz on the same 65nm technology generation. This higher clock frequency allows Harmoni to keep pace with high-performance processors that are running at clock frequencies of a few GHz and still have low overheads on performance.

## 1.4 Metadata caching

Runtime monitoring approaches that are built on the notion of tagging must also managed metadata stored in shared memory. Using memory bandwidth shared with the main processing core to access the metadata results in contention overheads that will be exacerbated by large metadata values, which may be needed for flexibility and functionality–desired characteristics of the co-processing architecture. In Chapter 5, we found that metadata for runtime monitoring approaches can exhibit the characteristics of sparsity, where many of the accesses are unnecessary accesses for blank metadata values; and frequent value locality, where only a few unique metadata values are actually used by a runtime monitoring function over a period of time.

To take advantage of these characteristics, we introduce two optimizations to improve the performance of on-chip metadata caches given a limited hardware budget. First, we showed how a default-value filter named Non-Default Metadata (NDM) cache can remove a large percentage of accesses (and misses) to the metadata caches. Next, we show how a dictionary-based compression approach named Dynamic Metadata Compression (DMC) can take advantage of frequent value locality to reduce accesses to memory without dramatically increasing the

size of the last-level metadata caches. Our evaluation of the combined approach showed that the optimizations can reduce metadata cache misses and reduce the overheads of runtime monitoring when large metadata values are used.

CHAPTER 2

# RELATED WORK

This chapter summarizes some of the related works, which can be broadly categorized into runtime monitoring schemes, generalized monitoring architectures, and cache compression. While this chapter does not entirely cover all of the work in these areas, it does highlight some of the main works and the trade-offs that were made between convenience, efficiency, and error coverage. For runtime monitoring schemes that address specific errors, we will also discuss how they can be implemented by the proposed co-processing architecture.

## 2.1 Runtime Monitoring Schemes

### 2.1.1 Integrity Checking

| Operation | Checker |
|---|---|
| Read Instructions | Control-flow Integrity |
| Compute | Computational Integrity |
| Read and Write Memory | Data-flow Integrity |

Table 2.1: Basic computing operations and integrity checkers

Modern computing systems are typically built on modified Harvard architectures. Programs that run on these machines require the correct execution of three basic classes of operations, which are shown in Table 2.1. The correct functionality of these architectures can be verified by checking that these basic operations have been correctly performed using a combination of operand data, metadata, and arithmetic operations for each. Integrity checking schemes have

been proposed to check these basic operations by attempting to infer a set of rules regarding programmer intentions and then using runtime checks to verify the adherence to these rules.

For instructions, control-flow integrity schemes [105, 1, 90, 91] statically analyze source code at compile time to build control-flow graphs (CFG) that represent valid executions of the program. Using the results of static analysis, control-flow integrity schemes can embed metadata in the form of labels in the basic blocks of an instrumented program. At runtime, these labels can be used to check whether basic block transitions correspond to valid edges in the statically-determined CFG for the application [90, 91, 105]. These checks allow control-flow integrity schemes to detect a wide-range of control-hijacking attacks [106], which can manifest as the transfer of control to attacker-injected code or privileged functions in standard libraries. However, control-flow integrity schemes are limited by the imprecision of static analysis, which must make approximations (such as context-insensitive or flow-insensitive analysis) that reduce precision so as to allow the analysis to complete in reasonable time and space. Hence, monitoring at the granularity of basic blocks can be too coarse: if static analysis tools are unable to identify basic blocks at a fine granularity, errors that cause a monitored application to jump into the middle of a basic block would be able to bypass the preceding checks.

For compute, computational integrity schemes have been proposed that use a small hardware checker to verify that the computation was performed correctly by the components of the larger processing core [10, 90]. For example, Argus [90] used a hardware checker that is tightly coupled to the functional units of a processing core to check computed results. Rather than duplicating

the functional units being checked, the hardware checkers proposed in Argus were simplified by only having to verify certain properties of the computation. As another example, DIVA [10] was proposed as a more comprehensive hardware checker for the processing core. The checker is attached to the commit stage of the processor and checks that committed instructions were performed correctly by the processing core. For permanent stuck-at faults, DIVA can act as a redundant hardware module that can assume the responsibilities of the processor's data-path. While the processor pipeline will operate with lower performance because of the slower DIVA checker, the system will still function correctly and provide graceful degradation instead of hard-stop failure behavior. In this work, we will demonstrate the functionality of the co-processing architecture by showing how a computational-integrity checker can be implemented on the co-processing fabric by building an expressive interface that can communicate operand values to the checker.

For memory reads and writes, data-flow integrity schemes [23, 3, 159] have been proposed that attempt to check that the values read by the processing core matches programmer intentions. DFI [23] statically analyzes the monitored application using points-to and reaching-definitions analysis to determine the set of instructions that will normally write each memory location and then instruments the program binary with checks for set membership on dynamic write instructions. These checks are facilitated at runtime by instrumented operations that use metadata to track the ID of the last instruction to write to each location. This data structure is then used by the runtime checking operations to confirm that the values read by consuming instructions were written by legal producing instructions. When evaluated with SPEC CPU 2000 benchmarks, DFI had very high performance (104%) and memory usage (50%) overheads. To reduce the

high performance overheads of DFI, WIT [3] compressed the encoding of instructions that are allowed to write to a particular memory location by pairing them using a small set of unique values (or colors). In a sense, WIT can detect illegal writes by checking that the write instruction's metadata, which encodes the color, matches the metadata of the target memory location that the instruction is attempting to write. However, similar to control-flow integrity, dataflow integrity schemes are still limited by the precision of static analysis in finding memory errors in the monitored application. The inaccuracy of static analysis also means that data-flow integrity schemes often require complementary mechanisms [3] to improve error coverage.

As a simplified form of data-flow integrity, memory integrity schemes [10, 62, 130, 166, 91] check that memory operations were correctly performed by the main processing core. DIVA [10] checks the correctness of memory operations by re-executing them. Argus [91] protects against errors that cause a load or store to access the wrong word by XORing the address and data and using parity to protect the XOR result. AEGIS [130] operated under the premise that malicious entities had physical access to a computing system and hence everything outside of the CPU die can be considered untrustworthy. AEGIS verified that data stored in memory were untampered with by using a tree-based integrity verification scheme to build a complete hash of memory contents.

Many of the essential elements for a general purpose monitoring platform can be discerned from a study of the related works in integrity checking. In particular, we will demonstrate how the co-processing platform presented in this thesis can make use of an expressive interface from the processing core to the checker, metadata for application data, and programmable hardware to imple-

ment many of the integrity-checking schemes presented in this section. Further, we will show how programmable hardware support can be leveraged to perform bookkeeping and checking operations at runtime and do so with performance overheads on monitored applications that are comparable to approaches that use custom hardware.

### 2.1.2 Return Address Protection

Buffer overflows are well-known software-errors [115] that continue to manifest in zero-day security exploits [14, 86, 162]. Buffer overflow errors occur when unsanitized program inputs are used by a vulnerable application to write beyond the boundaries of a program buffer. When the buffer is allocated in the program stack, the error can lead to the corruption of sensitive data adjacent to the buffer such as the function's return address. By overwriting the buffer with code or data of the attacker's choosing and corrupting the return address, a malicious attack can hijack control of the vulnerable program [106].

Because corrupting pointers such as the return address is essential to the success of typical buffer overflow attacks [131], many runtime monitoring techniques were developed to counter these attacks by protecting the return address. StackGuard [39] and ProPolice [68] proposed to modify application binaries by placing canaries next to the return address. When overflows occur with canaries, the canary would be corrupted (along with the return address), and the overflow can be detected by checking that the value of the canary was unaltered at a function return. However, canaries are an ad-hoc mechanism to protect against specific buffer overflow exploits that linearly overwrite adjacent

data [21]. Given details of the canary implementation, an attackers can also ”guess” the canary value and escape detection.

Other return address protection mechanisms attempted to better protect canary values. PointGuard [38] proposed to protect return addresses and function pointers by encrypting them when they are stored in memory. This approach reduces the probability of a successful attack, as attacks are no longer able to control the destination of function returns unless the encryption key was also known. Libsafe and Libverify [14] is a two-step approach that attempted to protect against buffer overflows and protect the return address. First, Libsafe intercepted unsafe C library functions for memory and string copies and checked that they cannot cause overflows that can reach the return address. Next, Libverify makes a backup copy of the return address on function entry and checks that the return address still matches the backup copy when the function returns. To prevent canary values from being ”guessed,” fine-grained memory protection mechanisms [64, 122, 101] can be used to implement canaries that trigger protection faults when they are written to. For example, fine-grained memory protection may be implemented using two-bits of metadata for each memory word to track allocated/initialized status; when the canary is marked unallocated, a malicious overwrite that writes to the canary would be detected as a return address protection error [64]. Despite the inherent limitations of canary-based approaches, they can still effectively protect specific data in the instrumented application. Further, we will demonstrate in the later chapters how such techniques can be also implemented on the co-processing architecture proposed in this thesis.

### 2.1.3 Information Flow

Despite the wealth of works available in protecting the return address, its corruption is just one of the many ways that a security exploit can manifest. Information flow tracking techniques aim to protect against a wider range of security exploits by targeting the final step, which is to coerce a compromised application to jump to code of the attacker's choosing. Also known as tainting, information flow tracking techniques function by tainting untrusted data and then checking that tainted data are not used as pointers in security sensitive operations [131, 138]. By exposing an interface to taint information to user-level applications, which can check the taint of arguments to sensitive function calls, information flow tracking can be used to detect even high level semantic errors such as command injection and cross-site scripting [103, 156, 43].

Tainting approaches built on software instrumentation frameworks [102, 122] can run unmodified program binaries but incur performance overheads of as much as 100X. Although as much as 40% of the overheads [156] of tainting could be attributed to the use of heavyweight binary instrumentation frameworks [101], even with aggressive optimizations to eliminate unnecessary or redundant checks, overheads for serial software-based approach are still often 100% or more [156, 113]. Alternatively, hardware-based taint tracking approaches have been proposed that offload tainting operations as much as possible using dedicated registers, memory, and computational resources for taint metadata [41, 131, 43]. In this thesis, we will also show how taint tracking can be mapped to a the proposed co-processing architecture and have average performance overheads of less than 10% on monitored applications.

One of the limitations of taint tracking is that it must make difficult trade-

offs between false positives and false negatives. Taint tracking approaches have been proposed that address tainted data pointer errors [131, 41, 26, 43]. However, these works must also take measures to stop the propagation of taint when branch conditions are also tainted or the spread of taint may explode [34], which leads to undesirable false positives. To reduce false positives, the program may need to be recompiled [131], or assumptions about programmer intentions [26] must be used to proactively untaint application data. However, attacks that corrupt data throw control dependencies may still be able to subvert the security of a target system [29] and lead to false negatives.

Taint tracking can be viewed as a simplified subset of information flow control that only allows for two levels of privilege: trustworthy or not [110]. In a broader sense, Information flow control techniques attempt to control the movement of data in a system. For example, multi-level security policies such as Bell-LaPadula (BLP) or Biba [18] use privilege levels and policies on the movement of data between privilege levels that prohibit the dissemination of information from low to high levels of privilege. Under such policies, when a subject reads an object that contains low-integrity information, then the privilege level of the subject is lowered to the minimum of the object's and its own. Once its privilege is lowered, the subject is no longer allowed to write to locations that require higher levels of privilege (including output channels such as the network interface), thus preventing the dissemination of the sensitive information it has read. Many system have been proposed that were built on similar policies by creating explicit labels for all data, programs, storage, and I/O channels of a system [144, 126, 160, 161]. In this thesis, we will also show how information flow control techniques can be implemented on the proposed co-processing architecture. Further, we will also show how the larger and more expressive metadata

values that are needed to encode the labels can be supported in an efficient manner.

### 2.1.4 Bounds Checking

One drawback of information flow tracking is that the checks take place after data has been corrupted, this necessitate fail-stop behavior on a detected error. Bounds checking techniques attempt to detect a similar class of errors but do so before sensitive data can become corrupted. This can allow application that require high availability to be unimpeded by the manifestation of such errors [114].

For speed and efficiency, C and C++ provide programmers the freedom to perform arbitrary arithmetic operations on pointers. This speed is reflected in the C standard library, where bulk copy operations (such as `strcpy()`, `strcat()`, `sprintf()`, and `gets()`) do not check that the destination buffers are large enough to hold the copied data. Programming mistakes in the use of these features can lead to buffer overflows [106]. To retrofit pointer safety, researcher have recognized that an important invariant of the C/C++ programming languages is that the intended referent (delimited by the memory address boundaries of the object) of each pointer are typically invariant over the lifetime of the pointer [63]. Bounds checking techniques build on this invariant and instrument the application at compile time to track the intended referent of each pointer and to check pointers when they are dereferenced. As is the case with other monitoring techniques discussed in this chapter, bounds checking techniques also must make difficult trade-offs and can be divided into two distinct

types.

Object-based bounds checking techniques track the base and bounds information of allocated objects in a centralized data structure [77, 117, 157, 47, 4]. Object-based referent checking techniques offer the advantages of being backwards compatible with the calling conventions of pre-existing library functions and being able to check for temporal memory errors. The seminal work proposed by Jones and Kelly [77] stored the base and bounds of live objects in a a splay tree that is looked up using the pointer's value. However, Jone's and Kelly's proposed approach had the drawbacks of high overheads on performance and false positives. Because C and C++ allow a pointer to temporarily move out-of-bounds without being dereferenced, as is the case in loop termination conditions, Jones and Kelly's approach suffered false positives by checking the result of pointer arithmetic operations. CRED [117] resolved the false positives in Jones and Kelly's approach by retaining information about out-of-bounds pointers in a separate data structure and checking pointers when they are de-referenced instead. When out-of-bounds pointers return in-bounds later in the program, the additional data structure proposed in CRED allows the pointer's bounds information to be recovered. In addition, previous works showed how the performance overheads of object-based bounds checking techniques can be reduced by restricting the scope of pointers that must be checked [48, 117, 13], or using static analysis to remove redundant checks [19, 48]. However, because objects in the centralized data structure is represented by its start address (and this start address may be shared with an array inside a struct), one of the main drawbacks of object-based approaches is that it is unable to detect sub-object overflows. This drawback renders object-based bounds checking approaches incomplete as solutions for memory safety.

In contrast to object-based bounds checking approaches, pointer-based bounds checking approaches [11, 108, 75, 99, 97, 33] semantically change pointers into objects that also carry base and bounds information about its intended referent. By allowing multiple pointers to be associated with the same object but have different bounds, "fat pointer" techniques are able to address the incompleteness of object-based bounds checking approaches. However, by increasing the size of pointer data, "fat pointer" techniques render the instrumented application incompatible with pre-existing library functions by changing the memory layout and the semantics of function calls. Some of this compatibility can be restored by mapping the bounds information for pointers to metadata [44], which makes them also more conducive to hardware implementation. Fat pointers can also be viewed as adding capabilities addressing to pointers, where pointers carry information regarding the address range that they can access and privileges they have on the data. The CHERI Capability Model [151] demonstrated how tagged memory and a MIPS co-processor that is attached to the compute and memory access stages of the processor pipeline can be used to implement capability-based addressing and complement the protections afforded by virtual memory. By using 1-bit of metadata to distinguish between normal data and capabilities and extensive compiler support to instrument applications with capabilities checking, CHERI is able to achieve fairly low performance overheads (20%) on monitored applications. In this thesis, we will show how a similar hardware bounds checking approach can be mapped to the co-processing architecture proposed in this work.

However, pointer-based bounds checking approaches also comes with drawbacks in terms of being unable to check for temporal errors. Unlike object-based referent checking approaches that can use its centralized data struc-

tures to revoke objects that are deallocated, pointer-based referent checking approaches such as Cyclone [75] and CCured [97] are only able to protect against temporal memory errors if a garbage collection mechanism is is available. This reliance on garbage collection makes the performance of a program unpredictable and negates some of the performance advantages of the C programming language.

### 2.1.5 Discussion

From the short introduction to some of the past works on runtime monitoring, it can be observed that runtime monitoring techniques are largely ad-hoc and have trade-offs that limit their use. The existing approaches have been shown to be effective in defending against known exploits, but they also have drawbacks stemming from the need to avoid false positives or having false negatives. Once an approach becomes more widely used and therefore scrutinized, their drawbacks can become avenues for future exploits to target. Hence, the set of important monitoring functions will likely change over time as new zero-day exploits emerge and protection mechanisms are adapted to defend against them.

## 2.2 Runtime Monitoring Architectures

### 2.2.1 System Call Interposition

For reusability, large systems are often built up from generic helper modules and third party libraries, which may contain buggy components or security vul-

nerabilities. To sandbox these and other components of an application, system call interposition frameworks were proposed as mechanisms that can check for anomalous behavior. In theory, these techniques can be an effective last line of defense as many software attacks can only effect lasting damage to the system through the use of system calls [54, 73, 111]. Further, these frameworks typically operate on the principle of least privilege, which permits programs to use only system resources that are necessary for its legitimate purpose.

Fraser et al [54] created a platform for system call interposition with mechanisms for tracking system call history and for reading and writing system call arguments and return values. Multiple works proposed spawning dedicated processes that perform the monitoring using process tracing facilities available in the operating system to intercept and interpose system calls [59, 73]. These approaches deny all system calls by default and use explicit white lists to determine what can be allowed. Systrace [111] generated the same white-listing policies in a more automated manner by running the target application with known-good inputs and recording the system calls that are needed.

Because system calls require a long-latency context switch, the overheads of the extra analysis that is performed by system call interposition techniques have a negligible impact on performance [59, 73]. However, the use of white lists limits system call interposition techniques to programs that do not legitimately need many privileges [59] and do not contain covert channels [57]. Further, the coarse-granularity of checking can result in very long latencies between the occurrence of an error and when it is detected. In contrast to the coarse granularity of system call interposition, the monitoring approaches that can be implemented by the co-processing architecture proposed in this thesis will fo-

cus on the implementation of monitoring techniques that can be invoked at a finer granularity of individual instructions. This fine granularity of invocation can provide for checkers that can capture software errors earlier and more accurately than coarse-grained approaches.

### 2.2.2 Multi-variant Systems

Multi-variant systems are general purpose execution frameworks that are built to detect errors in an application by concurrently running multiple variants of the same application in lockstep. The variants are designed to produce the same result given the same inputs, but an error in any one variant will cause it to deviate in behavior with regards to the other variants. For example, N-version programming [25, 12] proposed to have teams of developers, who do not interact during the programming process, independently generate functionally equivalent programs from the same specification. During runtime, all variants are blocked at specific points of the execution, for example at system calls [40, 119], so that a monitoring process can examine the intermediate states to detect behavioral deviations.

Multi-variant systems have been extended to monitoring for security errors by judiciously selecting the properties to change [72], which determine the types of security errors that can be detected. Orchestra [119] used automatic program transformation techniques to generate variants where the program stack grows in opposite directions [118] so as to detect stack smashing attacks. DieHard [16] used program stack and heap layout randomization techniques in different variants so as to probabilistically avoid buffer overflow and format string errors.

N-variant systems [40] leveraged a combination of memory layout randomization [17, 109, 116] and instruction set randomization [15, 79] to detect arbitrary memory overwrites and code injection errors. Other properties, such as system call randomization, register randomization, library entry point randomization, canaries [39], and padding between stack and heap variables [67] can also be leveraged in some combination to detect specific symptoms of security errors.

For security and reliability, multi-variant systems trade off performance, power, hardware, and software development costs. Multi-variant systems built on on multi-core clusters [25, 12] can greatly increase the power consumption and hardware costs to run an application. Further, the need to have multiple differing replicas can potentially increase the cost of software design [12]. In contrast to multi-variant systems, the co-processing architecture proposed in this thesis will implement checkers that can target specific aspects of the computation. This choice of design allows the checkers to be simpler and more efficient than those that are implemented by multi-variant systems.

### 2.2.3 Monitoring Platforms

Software-based runtime monitoring approaches such as those that are built on dynamic binary instrumentation (DBI) frameworks [70, 101] are effective in aiding the development of new monitoring schemes and being backwards compatible. However, the need to invoke instrumented software handlers that perform monitoring operations for each instruction of a monitored application can result in very high performance overheads for DBI-based approaches. Log-Based Architectures (LBA) [27] proposed to mitigate these overheads by taking advan-

tage of process-level parallelism. LBA proposed to communicate events in the monitored application to a separate processing core that can perform the monitoring operations in parallel. However, LBA incurs the full area and power overheads of another general-purpose processing core. Alternatively, Dynamic Instruction Stream Editing (DISE) [36] proposed to mitigate these overheads by taking advantage of instruction-level parallelism. DISE modified the decoder of a superscalar processing core such that it can be used to inject instructions that perform operations such as runtime monitoring into the dynamic instruction stream of a monitored application. While LBA and DISE are both able to avoid the performance overheads of DBI, they still incur high performance overheads (50% or more) on monitored applications. In this thesis, we will design and evaluate alternative approaches that have less area and power overheads than LBA, and lower performance overheads than both.

Other works have proposed to mitigate the performance overheads of runtime monitoring by using hardware support to memorize and filter software handler invocations. These filtering approaches can improve performance by handling previously-observed or common monitoring operations in hardware and maintain high flexibility by invoking the software handler in exceptional cases.

FlexiTaint [145] filtered software handlers that perform information flow tracking using a programmable Taint Propagation Cache (TPC) that was added to after the commit stage of the processing pipeline. The TPC memorizes the outcomes of previous invocations of software handlers and uses a combination of the input metadata and opcode to update the value of metadata that are 0 to 16-bits in size. For a small TPC cache of 4KB, the performance overheads

of information flow tracking were reduced to 3.7%. In contrast to FlexiTaint, we propose to build a more general purpose co-processing architecture that can support a wider variety of runtime monitoring approaches and do so with low performance overheads for each.

FADE and PUMP expanded on prior works (and the work in this thesis) by proposing general-purpose monitoring architectures that use rule caches to minimize software handler invocations. FADE [55] was proposed as a general purpose design for filtering monitoring events from a software handler. For a variety of runtime monitoring approaches, FADE showed that the average monitoring load consisted largely of redundant operations such as stack updates and rarely exceeded one instruction per cycle. Using a metadata cache and a rule table, FADE was shown to be able to filter out 84-99% of instructions and reduced the performance overheads on monitored applications to 20-80%. In the limit, hardware filtering approaches can be viewed as software-defined engines for metadata processing. While metadata can be accessed and metadata processing can be performed by hardware, the policies for metadata processing are still largely controlled by software, especially when metadata are pointers to larger data structures. Programmable Unit for Metadata Processing (PUMP) [45] was proposed as a way to implement such a software-defined engine. PUMP was a RISC co-processor with tagged memory, registers, and caches for metadata processing. PUMP utilizes rule caches that memorize that prior outcome of software handler invocations and dedicated metadata caches and memory for metadata propagation. To provide utmost generality, the rule caches in PUMP can map between the instruction opcode and up to five input tags to two output tags. To demonstrate the effectiveness of PUMP in implementing a wide range of monitoring approaches, example approaches such as information flow track-

ing, execute-disable, bounds checking, and control-flow integrity were built and evaluated on the co-processing architecture.

In this thesis, we will attempt to minimize performance overheads of runtime monitoring by mainly focusing on hardware techniques that can transparently perform bookkeeping and checks on the co-processing fabric. By taking advantage of the configurability of the co-processing fabric, these techniques can have the lowest overheads on performance. However, to retain flexibility and to increase performance, we also adapted a similar approach of invoking software handlers for complex monitoring operations and using a programmable tables to memorize the outcomes and filter redundant software handler invocations.

## 2.3 Cache Compression

In Chapter 5, we will study the effects of optimizations on the metadata cache hierarchy that were adapted from prior works in cache and memory compression. In this section, we describe some of these related works in cache and memory compression.

Previous research in runtime monitoring have shown that metadata for various types of runtime monitoring approaches have good spatial [131, 127, 137] or temporal locality [161, 145, 96]. Spatial locality, which refers to large chunks of contiguous application data having identical metadata, is a characteristic of metadata for information flow tracking [131, 127, 137]. Prior research has leveraged this characteristic to use one instance of metadata to represent a large chunk of application data in memory [131, 127] or in the on-chip metadata

caches [137]. Unlike small tags, which have higher spatial locality, we aimed to optimize for large tags, which have more permutations and thus lower spatial locality. In this work, we proposed NDM cache to take advantage of sparsity to filter accesses to the metadata caches. This makes NDM orthogonal and complementary to prior works such as the range cache [137], which aimed to optimized the storage of metadata in on-chip caches. In addition, temporal locality, which refers to few unique metadata values being utilized in each phase of application execution, is a characteristic that has been demonstrated for metadata managed by array bounds checking [96], information flow control [161], and information flow tracking [145]. Those works [96, 161, 145] took advantage of temporal locality to speedup otherwise slow checks by caching checks for the most frequently observed metadata values and reusing them for the same operation and operands. In this work, we found that metadata values exhibit excellent temporal locality. In contrast to previous works, which take advantage of temporal locality to memorize the outcome of metadata checks, we took advantage of temporal locality to optimize the storage of metadata values in on-chip caches.

Prior works have also evaluated compressing data stored in main memory. For lossless on-chip data compression, Dusser and Seznec [49] showed that null data blocks comprise a significant fraction of application data in memory, by dividing the address space into strictly null or non-null data, the null data blocks can be represented using only one bit of memory. Several authors [80, 30, 158, 6] observed that application data exhibit repeating patterns. By representing these patterns using fewer bits, they are able to increase compression ratios by two- to threefold. Other works [139, 24] proposed to take advantage of the statistical redundancy of data streams by using well-known compression algorithms such as LZW to compress data blocks that are stored in caches or memory. In

contrast to these works on main memory compression, we aimed to compress data stored in on-chip caches so as to make more effective use of limited on-chip storage and off-chip bandwidth.

However, Shannon's source coding theorem also indicates that not all data can be compressed. Given that uncompressible data may actually exacerbate data bandwidth and storage requirements of cache compression techniques, Lee [84, 85] and Almadeen [5] evaluated systems where compression is adaptively enabled depending on whether it benefits performance. In this chapter, we simplified the problem of storing both uncompressed data from compressed data by partitioning them into distinct halves. Other proposed approaches can similarly reuse the same storage medium for both types of data, at the cost of lower compression ratios. For example, Yang et al [158] proposed a compressed cache design that allocates additional tags for cache lines such that they can either hold one uncompressed cache line or two cache lines that have been compressed to at least half their original length. This bounds the compression ratio to at most two. Similar to the approach in this work, Chen et al [24] proposed to dynamically partition the L2 cache into sections of different compressibilities. An attempt is made to compress each cache line that is brought into the L2 cache, and the line is placed into the relevant partition depending on the compression outcome. However, such rigid partitions can lead to internal fragmentation when data cannot be exactly compressed into distinct granularities. Indirect-index caches [123, 61, 120] were presented as a possible solution by allowing tags to be decoupled from data via a level of indirection, at the cost of longer access delays because they can no longer occur in parallel.

CHAPTER 3

# FLEXIBLE AND EFFICIENT RUN-TIME MONITORING ON RECONFIGURABLE HARDWARE

## 3.1 Introduction

In this chapter, we present an architectural framework, named FlexCore, that features a general processing core augmented with an on-chip accelerator fabric. The accelerator fabric in this framework is composed of bit-level reconfigurable logic that's organized similarly to typical island-style FPGAs. The fabric can be configured for a hardware implementation of logic for any computation that can fit inside the fabric. FlexCore is designed to enable a large class of run-time monitoring techniques to run efficiently and in a transparent fashion for applications running on processing core. A new runtime monitoring technique, even if it was not known at the time the processor was fabricated, can be implemented in hardware on the accelerator fabric and run with greater efficiency than purely software approaches. The reconfigurable fabric also enables the monitoring functions to be customized for each processor instance and each application.

The design of the FlexCore architecture was motivated by the fact that existing proposals for run-time monitoring suffer from drawbacks of either high overheads or low flexibility. Software-based approaches in runtime monitoring leverage the inherent programmability of general purpose processors to implement monitoring operations in software by instrumenting each instruction of the monitored application with several additional instructions for bookkeeping and checking [102]. Software-based approaches offer very high flexibility

in allowing monitoring functions to be modified long after the chip has been fabricated, however, this flexibility comes at the cost of drastically reduced performance for the monitored application. A software implementation for DIFT monitoring on a single core is reported to have an average slowdown of 24X [102], and even with aggressive optimizations, this overhead is still as high as 3.6X [113]. Running the monitoring operations on another processing core of a CMP [125, 28, 162] can lower the performance overheads but this is rather inefficient as it requires more than twice the power. Further, a general purpose processing core can be a poor match for run-time monitoring schemes that do not need to fully replicate the computation [10] or just perform simple checks [131]. As an example, dynamic information flow tracking [131, 41, 43] has primary operations of propagating and checking the tag of the operands used by a subset of instructions executed by the monitored application. These checks can be performed using logical operations for just a few bits; a full 32/64-bit general purpose processing pipeline would be over-provisioned in terms of cost and power to perform these operations. On the other hand, specialized hardware techniques [131, 146, 145] are the most efficient for implementing run-time monitoring operations, but are inflexible and cannot adapt to changes in its environment.

In the FlexCore architecture, we propose to add an on-chip reconfigurable fabric to a traditional processing core to create a highly flexible and efficient runtime monitoring platform. We believe that a bit-level reconfigurable fabric can be better suited for run-time monitoring compared to software running on traditional processing cores as many monitoring techniques for security, reliability, and programmability require mostly bit-level logical operations. Further, the fabric can be dynamically reconfigured to implement the logic of any mon-

itoring function that fits within the fabric, giving the system the flexibility it needs to response to changes in its usage or environment. Hence, the FlexCore architecture can provide a solution that bridges the tradeoffs made between specialized hardware and software instrumentation-based approaches for runtime monitoring.

However, the inefficiency of the reconfigurable fabric compared to custom logic was a problem that needed to be overcome. A circuit that is implemented on a FPGA can have much longer critical path delays than a custom implementation [81]. Therefore, we needed to identify techniques to hide the inefficiencies of the reconfigurable fabric and to increase its throughput so that it would not impede the progress of the general purpose processing core. To accomplish this goal, we used a small instruction queue between the general purpose processing core to decouple operations on accelerator fabric from the monitored application. Next, we added control registers to the main processing core so that instructions that are not relevant to the monitoring operation being performed are filtered out and not forwarded to the accelerator fabric. Finally, we selectively incorporated custom logic in the reconfigurable fabric to perform common operations that are inefficient to implement on a bit-level reconfigurable fabric.

To evaluate the proposed architecture, we built a prototype of FlexCore based on a simple in-order microprocessor (Leon3) and studied the area, power consumption, and performance on 65nm process technology. Evaluation results of the prototype demonstrate that the FlexCore architecture can indeed support a range of runtime monitoring functions with different requirements and operations, including functions that protect against uninitialized memory reads,

buffer overflow, control hijacking, and soft errors in the processor's data-path. Results from a study using synthesis tools showed that the interfaces added to the main processing core to support communication with the reconfigurable fabric have minimal impact on the operating frequency of the main computing core. In terms of the silicon area, FlexCore adds about $0.3mm^2$ for dedicated hardware components and all evaluated extensions can fit in a $0.4mm^2$ FPGA fabric, which represents a small increase for modern processors that are tens of $mm^2$.

The experimental results also suggest that run-time monitoring on FlexCore is far more efficient than software implementations in terms of both performance and power consumption, and can almost match the performance of ASIC implementations for processors with moderate frequencies. For example, array bounds checks can be performed by the reconfigurable fabric with a 18% average slowdown and 22% additional power consumption while an ASIC implementation results in a 8% slowdown with 8% additional power consumption. Dynamic Information Flow Tracking (DIFT) on the FlexCore design incurs a 17% slowdown with a 21% power overhead compared to the 5% and 6% overheads of an ASIC. Overall, the prototype implementation shows that the FlexCore architecture is promising direction for the creation of a platform for flexible and efficient implementation of hardware runtime monitoring features.

The rest of this chapter is organized as follows. Section 3.2 describes the co-processing model and show how example monitoring functions can map to the accelerator fabric. Section 3.3 presents the design of the FlexCore architecture. Section 3.4 describes the details of our prototype implementations and how example runtime monitoring functions map to the fabric. Section 3.5 studies the

performance, area, and power consumption of our prototype runtime monitoring functions in both FlexCore and full ASIC implementations.

## 3.2 Co-Processing Model for Run-Time Monitoring

This section presents the computation model used by FlexCore for the fine-grained run-time monitoring. In the ensuing discussions, we will use the term "co-processor" to refer to the hardware implementation of a particular runtime monitoring function.

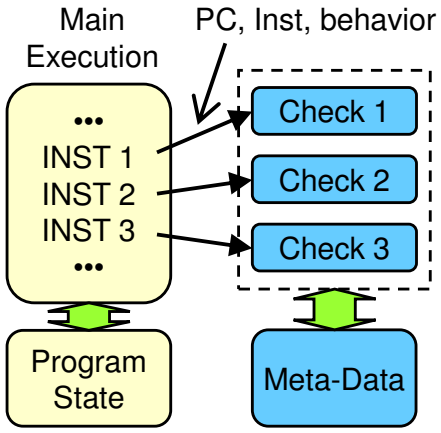



Figure 3.1: Computation model for fine-grained runtime monitoring

Figure 3.1 shows the high-level model of the computational model used by FlexCore. In the figure, dark (blue) blocks represent the co-processor and light (yellow) blocks represent the monitored computation. Conceptually, the co-processor performs fine-grained monitoring by observing each instruction that is executed by the monitored application and performing one or more operations for each instruction. In terms of operations, the co-processor can perform a combination of performing bookkeeping and checking for invariants of the

computation. For bookkeeping, the co-processor maintains meta-data that is often be disjoint from the program state to allow it to acutely track the properties of the computation. For invariant checking, if a check fails, the co-processor will have provisions to allow it to communicate to the main processing core that a problem has occurred. From computational model, it can be observed a runtime monitoring function can be characterized at a high level by its meta-data, instruction-specific operations, and software interfaces. Here, we briefly define each characteristics and its implication for the FlexCore design.

- *Meta-data*: Runtime monitoring functions may need meta-data for all computational state that is affected by the monitored application. For traditional cores, computational state is typically held and communicated through architectural registers and addressable memory. Hence, the co-processor needs to be able to have the capability to maintain metadata for each register on the main processing core and for each minimally addressable memory location used by the monitored application.
- *Operations*: One or more *fine-grained* or per-instructions operations are possibly necessary for performing the bookkeeping and checking operations required by the monitoring function. The operations performed will vary between different types of runtime monitoring functions, therefore the accelerator fabric must be flexible in being able to support the various types of computation that is required by the operations. Aside from decoded properties of the executed instruction, there is typically no logical dependence between data computed by the monitored application and meta-data maintained by the runtime monitoring function. Hence, the runtime monitoring operations can be *decoupled* from the main computation and be performed concurrently with subsequent instructions in the monitored

application. However, if a check does fail, then the hardware must have the ability to undo the effects of the instruction that triggered the failed check.

- *Software interfaces*: The runtime monitoring functions need to be able to communicate with the main processing core the results of certain checks. Simultaneously, the main processing core will need an interface to communicate completed instructions and some amount of decoded operands and relevant processor state to the monitoring function. In addition, the main processing core, in privileged mode, will need to be able to control and program the reconfigurable accelerator fabric. However, we anticipate that these programming and control interfaces will be used infrequently and need not be exceptionally expedient.

### 3.2.1 Example Monitoring Extensions

To further flesh out details of the processing model, we studied past works in hardware runtime monitoring and identified a subset of example monitoring functions as candidates for implementation on the proposed FlexCore platform. This subsection will show how those example runtime monitoring functions map to FlexCore's co-processing architecture.

Table 3.1 summarizes the operations of four example extensions: UMC, DIFT, BC, and SEC. Section 3.4 describes more details of each extension and its implementation.

| Extension | Meta-Data | Transparent Operations | SW Visible Operations |
|---|---|---|---|
| UMC [64] | 1-bit per MEM word | Set metadata on a store<br>Check metadata on a load | Clear REG/MEM metadata<br>Exception when a check fails |
| DIFT [131] | 1-bit per REG<br>1-bit per MEM word | Propagate metadata on ALU/LD/ST<br>Check metadata on BR/JMP | Set REG/MEM metadata<br>Clear REG/MEM metadata<br>Set registers on co-processor<br>Exception when a check fails |
| BC [33] | 4-bits per REG<br>8-bits per MEM word | Propagate metadata on ALU/LD/ST<br>Check REG metadata against<br>the MEM metadata on a load/store | Set REG/MEM metadata<br>Clear REG/MEM metadata<br>Exception when a check fails |
| SEC [10, 90] | | Check ALU results | Exception when a check fails |

Table 3.1: Example FlexCore runtime monitoring functions. UMC: Uninitialized Memory Check. DIFT: Dynamic Information Flow Tracking, BC: Array Bound Checking, SEC: Soft Error Checking.

**Uninitialized Memory Checking (UMC)** Uninitialized Memory Check (UMC) is an approach based on HeapMon [125], which was proposed as a way to detect programming mistakes where a program variable is not initialized before use. In our implementation of UMC, the co-processor maintains a one-bit metadata for each byte of memory in the monitored application to track the two possible states that the memory byte could be in: *0-Uninitialized* or *1-Initialized*. The main processing core uses new instructions added to the ISA for writing to FlexCore metadata will initialize (set to *0*) the metadata of each byte of memory that is allocated by functions such as malloc() or deallocated on a call to free(). Transparently on each store instruction executed by the monitored application, the co-processor will set the metadata for each memory byte that is written to to an *Initialized* state. On each load instruction, the co-processor will check that the metadata of each memory byte that is loaded does not have metadata corresponding to an *Uninitialized* state.

**Dynamic Information Flow Tracking (DIFT)** Dynamic Information Flow Tracking DIFT) is based on previous works [131, 41, 102, 43] that detect errors or

exploits that take advantage of software vulnerabilities such as buffer overflow, format string, or others to corrupt control pointers in the monitored application [43]. DIFT functions by tracking the propagation of values derived from untrustworthy I/O channels and checking that these values do not become the operands of of instructions that influence the control flow of the monitored application. In our implementation of DIFT, the co-processor maintains a one-bit metadata for each architectural register and word of addressed memory to indicate the level of trustworthiness of the value stored in that location: the bit has a value of *0-Trusted* or *1-Untrusted*. Privileged modules in the operating system for networking will use new instructions added to the ISA for writing to FlexCore metadata to set the metadata of each word of memory read in from an untrustworthy network channel to *1*. For each arithmetic and logic (ALU), load (LD), or store (ST) instruction that is executed by the main processing core, the co-processor will update the metadata of the destination operand according to the metadata propagation rules that the hardware is configured to perform for that instruction type. When security critical instructions are executed by the monitored application, the co-processor checks that the metadata of each of its source operands are not *Untrusted*.

**Array Bounds Checking (BC)** Array bounds checking (BC) is based on previous works that perform reference checking using coloring [3, 33]. In contrast to approaches that store reference information using a centralized table of objects [117, 47, 96, 44] with their base and bounds information, BC is a simpler approach that assigns just enough ”colors” or unique IDs to the allocated memory of an application so that out-of-bounds writes can be detected by a pointer dereference writing to memory that does not match the pointer’s color. To im-

plement BC, the co-processor maintains four-bits of metadata for each register and eight-bits of metadata for each word of memory. The metadata in each register and the four LSBs of the metadata for each memory word in the monitored application are *COLOR1*; the four MSBs of metadata for each memory word of the monitored application are *COLOR2*. Compiler instrumentation is needed to insert FlexCore metadata write instructions to set the metadata of each pointer and memory block returned by memory allocation operations such as malloc() such that *COLOR1* of the pointer is equal to *COLOR2* of the allocated memory range. On each load instruction executed by the monitored application, the co-processor will load *COLOR1* of the accessed memory location to the destination metadata register. For instructions executed by the monitored application that perform pointer arithmetic, the co-processor will update the metadata of the destination operand according to custom metadata propagation rules for pointer metadata [33]. Lastly, on all memory access operations (including stores), the co-processor will also load *COLOR2* of the accessed memory location and confirm that *COLOR1* of the register operands used to calculate the memory address matches the loaded *COLOR2*.

**Soft Error Checking (SEC)** Soft error checking (SEC) is based on previous works that detect transient errors, such as bit-flips, in the processor's datapath [10, 90]. In our implementation of SEC, the co-processor maintains no metadata but verifies the computation using a simpler logical operation using partially decoded information about the instruction and its operands to check computed result. To implement SEC, the interface between the main processing core and the co-processor was expanded to including decoded information about the instruction type, the value of the instruction's operands, and the result of

the computation. For each instruction communicated from the main processing core, the co-processor computes an expected checksum for what is expected of the result based on the instruction type and its operands and a result checksum using the result communicated from the main processing core. At the final step of each monitoring operation, the expected checksum is checked against the result checksum and an error is communicated back to the main processing core if the checksums do not match.

**Other Monitoring Approaches** Besides the examples that were discussed in the preceding paragraphs, we believe that the FlexCore co-processing model is composed of building blocks that allow it to be used to implement a large class of other hardware runtime monitoring approaches. For example, the wealth of works in security and reliability that take the approach of explicitly assigning metadata to memory locations and then using customized checks using that metadata can all be easily mapped onto the FlexCore accelerator fabric. These approaches can provide functionality such as fine-grained memory protection [149], information flow control [144, 126, 160, 161], memory debugging support [166], and program checkpointing [155]. By communicating low-level details about the computation being performed by the main processing core, the co-processing model can perform event counting. The co-processing model can also use the interface to offload functionality from the main processing core and provide for features that facilitate garbage collection [76], computational acceleration [65], and more.

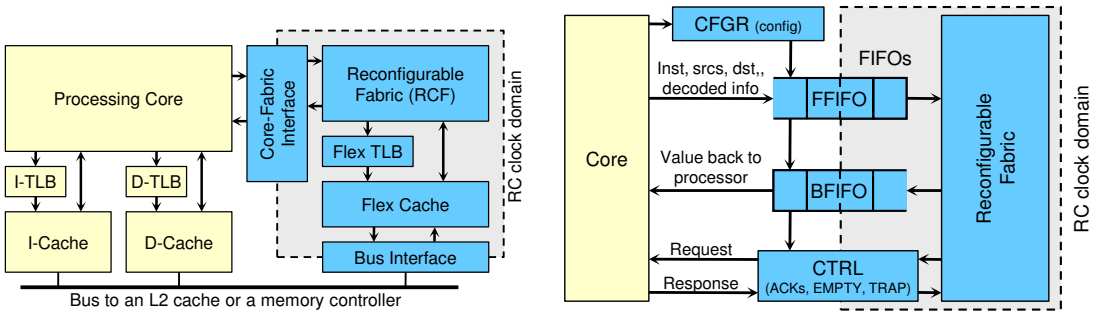
## 3.3 Architecture Overview

### 3.3.1 Scope and Design Goals

The following list summarizes our main goals in designing the high level architecture:

- *Flexible*: The FlexCore architecture should be flexible in having the capacity to map a broad range of runtime monitoring functions. The architecture should also be dynamically reconfigurable so as to allow new runtime monitoring functions to be added in the field.
- *Efficient*: Runtime monitoring functions on the FlexCore framework should be more efficient than software-only implementations, particularly in terms of power and performance.
- *Non-Intrusive*: To minimize the implementation costs, the architecture should only require minimal changes to the main processing core. Further, the added programmability should not degrade the performance of the main processing core.

Figure 3.2(a) shows the high-level block diagram of the FlexCore architecture. The yellow (light) rectangles represent components in traditional microprocessors and the blue (dark) rectangles represent new components for FlexCore. At a high-level, the architecture resembles the co-processing model that is described in the previous section. The main processing core forwards each completed instruction from the monitored application to the co-processor for runtime monitoring and receives signals from the co-processor through the Core-




(a) Overview (b) Core-fabric interface

Figure 3.2: FlexCore architecture block diagrams.

Fabric interface. The architecture provides a separate cache subsystem for meta-data with a separate cache hierarchy and metadata TLB for the co-processor. For cost and efficiency, the co-processor shares an interface to main memory with the main processing core.

The FlexCore architecture is carefully designed to exploit the common characteristics of run-time monitoring techniques without restricting the monitoring operations that can be performed. For example, unlike traditional FPGA co-processors, the main processing core forwards its execution trace without explicit intervention from software. The accelerator fabric is optimized for *bit operations*, which makes them a good match for run-time monitoring, and the meta-data cache hierarchy supports configurable read and write granularities of sizes that are a power of two between 1 and 64bits.

While the fine-grained reconfigurable fabric is a good fit for typical meta-data computations in run-time monitoring schemes, the bit-level reconfigurable fabric is a poor match for coarse-grained structures such as memory arrays and decoders. The FlexCore architecture handles inefficiencies in fine-grained reconfigurable fabric with a set of custom hardware optimizations.

- *Decoupled execution*: The co-processor is decoupled from the main processing core through the use of FIFO queues that enable non-blocking communication of instructions. In each cycle, the processing core places instructions that are guaranteed to commit into the FIFO. Unless the FIFO is full, the processing core can make progress independent of the reconfigurable fabric. This decoupling can hide the latencies of runtime monitoring operations for each instruction. Furthermore, it also allows the main processing core to run on its own clock domain and at speed that is independent of the reconfigurable fabric's maximum speed for a particular logic implementation. The clock domains are also carefully organized so that TLB and cache accesses by the both main processing core and the co-processor do not require more cross-domain communication.

- *Filtering and pre-processing*: The core-fabric interface can mitigate effects of the lower throughput of the reconfigurable fabric by only sending instructions that are relevant to the runtime monitoring function being performed on the accelerator fabric and filtering out the rest. Further, we also send partially decoded information about the instruction through the core-fabric interface so as to avoid the need to perform decode operations inefficiently on the reconfigurable fabric.

- *Specialized hardware modules*: The reconfigurable fabric incorporates a set of dedicated hardware units for functions that are common across many runtime monitoring functions and are expensive in terms of the performance penalty when mapped to the accelerator fabric. These modules include additional decoding logic, core-fabric interface, TLB, cache, and meta-data register file.

### 3.3.2 Core-Fabric Interface

| Function | Module | Field | Description | Bits |
|---|---|---|---|---|
| Config | CFGR | FFIFO | Select a FIFO behavior for each instruction type: 1) ignore, 2) accept only if not full, 3) accept and proceed, 4) accept and wait for an acknowledgement. Contains 2 bits for each of the main 32 instruction types (SPARC prototype). | 64 |
| Core To Fabric | CTRL | PACK | Acknowledgement for a trap signal from the co-processor. | 1 |
| | FFIFO | PC | Program counter. | 32 |
| | | INST | Undecoded instruction. | 32 |
| | | ADDR | Address for a load/store. | 32 |
| | | RES | Result of an instruction. | 32 |
| | | SRCV1 | Source operand 1 value. | 32 |
| | | SRCV2 | Source operand 2 value. | 32 |
| | | COND | Condition codes that affect instruction processing. | 4 |
| | | BRANCH | Computed branch direction information. | 1 |
| | | OPCODE | Decoded instruction opcode. | 5 |
| | | DECODE | Miscellaneous decoded signals. | 32 |
| | | EXTRA | Extra processor control signals. | 32 |
| | | SRC1 | Decoded Source1 register number. | 9 |
| | | SRC2 | Decoded Source2 register number. | 9 |
| | | DEST | Decoded Destination register number. | 9 |
| Fabric To Core | CTRL | CACK | Acknowledgement for FFIFO. | 1 |
| | | EMPTY | A signal to indicate that there is no pending instruction in the co-processor. | 1 |
| | | TRAP | Raise en exception. | 1 |
| | BFIFO | VAL | A return value on a 'read from co-processor' instruction. | 32 |

Table 3.2: The FlexCore interface between the core and the fabric.

The reconfigurable fabric communicates with the main core through a set of FIFO interfaces as shown in Figure 3.2(b). The FIFOs are connected to the commit stage of the main core. Table 3.2 lists in more detail the signals on the core-to-fabric interface. The core-to-fabric interface works to enable fine-grained instruction communication between the core and reconfigurable fabric. The processing core sends its execution trace to the co-processor using the FIFO interface, so that the fabric can perform monitoring or bookkeeping operations on each forwarded instruction.

A forward FIFO sends a trace of instructions, which are completed and ready to commit, in the program order. Each FIFO entry contains comprehensive in-

formation regarding the committed instruction, including a program counter, source and destination operands, ALU results, condition codes, and the branch outcome. In addition, we found that bit-level reconfigurable fabrics were a particularly poor fit for performing the decoding operations of a traditional processing core. Bit-level reconfigurable fabrics based on 4-bit or 6-bit CLBs expended 70-80% area and delay on programmable routing [74], this makes them a poor fit for performing complex decoding operations with very large fan-out. Hence, each FIFO entry also includes pre-decoded instruction fields such as an opcode, source register numbers, and a destination register number, in order to avoid performing instruction decode on the accelerator fabric. For comparison, we found that our DIFT prototype can run 30% faster by performing the instruction decoding for operands and control signals on the processing core.

A forwarding configuration register (CFGR) specifies how the forward FIFO handles each instruction type[1]. For example, the CFGR can be configured to forward load/store instructions but not ALU or control instructions when implementing UMC. The FIFO may also be configured to either allow an instruction to commit as soon as it is enqueued or stall the commit until there is also an acknowledgment from the co-processor that it is error free.

The reconfigurable fabric uses additional FIFOs to communicate back to the main core. A back FIFO (BFIFO) returns values from the co-processor for operations such as a special "read from co-processor" instruction added to the main processing core. In addition to data, the control module (CTRL) allows a set of synchronization operations between main core and the co-processor. The co-processor sends an acknowledgment back (CACK) for an instruction when the

[1]We augmented the decoding logic on the main processing core to drive an additional signal indicating what type (of 32) of instruction each is in the SPARC ISA

commit stage in the main core waits for a completion of an operation on the co-processor. In addition, the fabric provides a signal (EMPTY) to indicate whether there are any pending instructions in the co-processor. The reconfigurable fabric can also raise an exception using the trap signal (TRAP). If the interrupt level of the processor is sufficiently low, the main core acknowledges such an exception (PACK) and invokes a proper handler.

Note that the proposed FIFO interface with the reconfigurable accelerator fabric can easily support custom instructions on the main core for monitoring function. For example, in order to implement an instruction to set a configuration register within the reconfigurable fabric, the fabric can be programmed to update the register on a an instruction that would otherwise be a NOP for the main processing core.

### 3.3.3 Reconfigurable Fabric Architecture

The high-level FlexCore architecture is independent from the micro-architecture of the reconfigurable fabric and is applicable to various types of fabrics. In our evaluation of the FlexCore architecture, we used a standard LUT-based FPGA architecture with island-style interconnect that is similar to the Xilinx Virtex-5 [154], which includes standard Configurable Logic Blocks (CLBs) with LUTs and flip-flops. We used Xilinx ISE 12.4 [153] for the Xilinx Virtex V to evaluate the area, performance, and power consumption of our framework. The FPGA fabric is chosen over other coarse-grained fabrics because the bit-level reconfigurable fabric appeared to be a good fit for typical runtime monitoring functions that perform bit-level operations.

Our reconfigurable fabric also includes an embedded meta-data register file, which is implemented with custom hardware and has an 8-bit shadow register for each general-purpose architecture register in the main core. The register file provides an efficient way to keep meta-data for registers in a similar way that SRAM banks in commercial FPGAs provide efficient memory blocks. Because we already provide dedicated modules such as FIFO interfaces with decoded instructions, a TLB, and a cache for common co-processing operations, our accelerator fabric only includes standard Configurable Logic Blocks (CLBs) with LUTs and flip-flops without specialized dedicated modules such as block RAMs or DSP units.

### 3.3.4 Meta-Data Memory Hierarchy

For meta-data used by the reconfigurable fabric for bookkeeping, the reconfigurable fabric uses its own cache subsystem that is separate from the main core's L1 caches. This design minimizes interference with the main core's cache structures. Both the processing core and the reconfigurable fabric share the lower-level memory hierarchy such as an L2 cache and main memory. Currently, the architecture does not maintain coherency between the main core's L1 caches and the meta-data L1 cache. For the runtime monitoring functions that we studied, the co-processor only need to access meta-data in memory regions disjoint from program instructions and data. Cache coherence mechanisms can be added for future extensions that require fine-grained memory sharing between the main core and the co-processor.

The meta-data cache is almost identical to regular data caches except for the

capability to write at a bit granularity. Meta-data cache reads return the same 32-bit words as the cache in the baseline processor. For writes, the meta-data cache utilizes 32-bit write enable mask in addition to the address and the data word, and only bits within the cache word where the bit mask is set are updated. We found that the bit-level write capability is essential for efficient co-processing since many co-processing techniques work on meta-data much smaller than a word. Without this feature, a co-processor would need to perform two separate cache operations, a cache read followed by a cache write, in order to properly update the meta-data.

### 3.3.5 Programming Reconfigurable Fabric

The FlexCore architecture is designed for scenarios where the co-processor is treated as a hardware extension that only changes infrequently. For example, we envision that the reconfigurable fabric is programmed once and utilized until a long-running program has completed. In this context, the programming of the fabric does not need to be fast. Further, it makes sense from a cost and density perspective to store programming configuration bits in SRAM since this cost-effective approach dominates the market. Standard programming methods in today's commercial FPGAs, where a bitstream is serially shifted in to configuration memory, can be used to populate the configuration bits.

From a security and privacy perspective, because the reconfigurable fabric can closely monitor virtually all computations of the main core, its configuration memory must be protected from soft errors, and its programming must be restricted to only trusted parties. For security and privacy, a processor vendor

can choose from two options depending on how open the FlexCore feature is desired to be. First, the FlexCore programming can be treated similar to today's microcode updates, where the vendor can tightly control what updates can be performed. Only the vendor can create a valid update for the reconfigurable fabric and the programming interface is invisible to the software layer. Alternatively, the FlexCore programming interface can be exposed to an operating system, where only one process or processes with the highest privilege levels are allowed to update the reconfigurable. In both cases, the operating system must properly manage the meta-data memory space for monitoring functions that make use of metadata and ensure memory isolation between the meta-data of each process. Lastly, while we did not implement this, FlexCore could be further protected by making use of previously proposed techniques [7] of using CRC and error detecting codes to protect the configuration memory.

### 3.3.6 Precise Exception Support

To hide the latency of runtime monitoring operations, the FlexCore architecture decouples monitoring operations from the main processing core using a Core-Fabric FIFO queue and allowing the main processing core to push committed instructions into the queue and continue without waiting for monitoring operations for that instruction to complete. In this decoupled fashion, the main processing core will only stall the commit of completed instructions when the Core-Fabric queue is full, and this greatly improves the performance overheads of runtime monitoring on the FlexCore architecture. However, the decoupling also implies that the detection of an error in the co-processor may take place many cycles after the error has already occurred and modified system state, we

call this mode of error detection decoupled checking. In decoupled checking, by the time that an error is detected, additional instructions will complete on the main core, and more damage to the monitored application and state state could take place.

Runtime monitoring functions typically perform two types of operations: bookkeeping to update its metadata to reflect that of the monitored application; and checking using application data and metadata to ensure that an error did not occur. For bookkeeping operation performed by many runtime monitoring approaches, the bookkeeping can be performed in a decoupled fashion. Similarly, for runtime monitoring functions where damage from errors can be contained within the memory space of the monitored application and the application exhibits fail-stop behavior, the bookkeeping and checking operations can both be decoupled. However, for the remaining classes of applications and runtime monitoring functions, where errors could result in damage that is difficult to undo or recover from, then a mechanism needs to exist to ensure that critical instructions are checked before they are allowed to update system state, we call this mode of error detection precise checking. In the remainder of this subsection, we will discuss alternatives for implementing precise checking on different types of processing cores.

For security critical instructions and system calls where the monitored application may make irreversible changes to system system, these instructions can be forwarded to the runtime monitoring function but be delayed from commit until checks are complete. For modern out-of-order processors, such delays simply mean that instructions stay in an ROB (Re-Order Buffer) longer without necessarily stalling the execution of subsequent instructions. This simple

and straightforward approach for precise checking will have negligible effects on the performance of the monitored application when the frequency of critical instructions and system calls are low. However, if the frequency is high, then alternative mechanisms may be needed to mitigate the performance impact of such checks. We will discuss options for precise checking for out-of-order and in-order processing cores in the subsequent paragraphs.

For high-performance processing cores that make use of a ROB to reorder instructions and enable speculation, these architectures buffer instructions in its reorder buffer until the instructions are guaranteed to complete and are no longer speculative. For such architectures, we can simply extend the speculation support to cover instructions that have yet to be checked by the co-processor. Hence, instructions executed by the monitored application can complete and be readied for commit but not write to architectural state until the co-processor completes its checks.

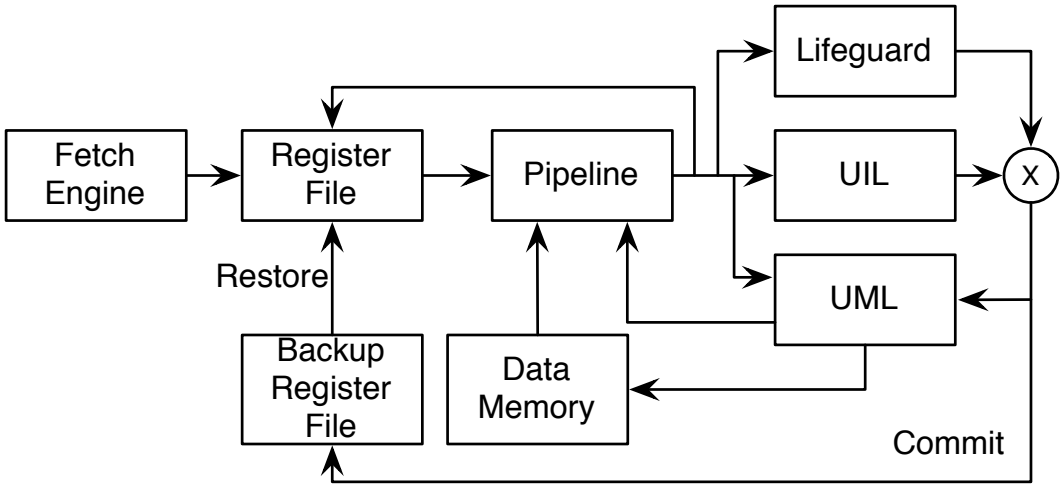



Figure 3.3: Block diagram of processing core with precise exception support.

For in-order processing cores that eschew speculation for area and energy efficiency, additional cost-effective hardware mechanisms are needed to allow for precise checking. Figure 3.3 shows the block diagram of one possible way that precise checking can be implemented on such architectures. The main idea behind the mechanism proposed in the figure is to ”buffer” completed instruction results so that the monitored application can continue to make forward progress and to allow each ”buffered” instruction to modify architectural state only when it does not result in an error. Intuitively, the approach can mitigate the overheads of checking when monitored applications exhibit bursty instruction execution patterns: the burst of commits can be ”buffered” and then checked, and the buffers can drain when the pipeline is idle and waiting for long latency operations, such as memory loads, to return. Rather than make extensive modifications to the in-order processing core, we leave the majority of the core unchanged. All instruction executed on the main processing core are allowed to update the register file in the main core pipeline immediately upon commit so that following instruction can use the results. However, memory writes are buffered in case a check fails and a copy of architectural state is preserved that can be restored on a detected error. To this end, we extend the architecture with several ”buffers” for completed instructions and their register and memory updates. The Unchecked Memory List (UML) buffers and bypasses speculative memory updates (stores) and each UML entry holds the value and the address of unchecked memory write. All loads on the main processing core are extended to check the UML for a matching entry before reading from the memory hierarchy. A Back-up Register File (BRF), which has the same number of entries as the main (speculative) register file, is added and is only written to by instructions that have been checked. The Unchecked Instruction List (UIL) is

a FIFO that holds a list of speculative instructions that are completed in the main core but not checked by the co-processor. The main core enqueues instructions into the UIL after completing an instruction and forwarding the instruction to the co-processor. Each entry in the UIL holds a pointer to an UML entry, destination register, and register write result.

As the co-processor completes the checking of each instruction, if the check passes, the next UIL entry is dequeued and its register and/or memory write are allowed to proceed to the BRF and/or memory. However, if the check fails, then the main core can restores its state to the before the failing instruction using the additional "buffers". The register file can be restored by copying values from the BRF. Next, the core flushes all entries in the UIL and UML and the pipeline, sets the appropriate status registers to indicate that a failed check occurred, and a control transfer instruction to the base address of the exception handler can be inserted into the processor pipeline.

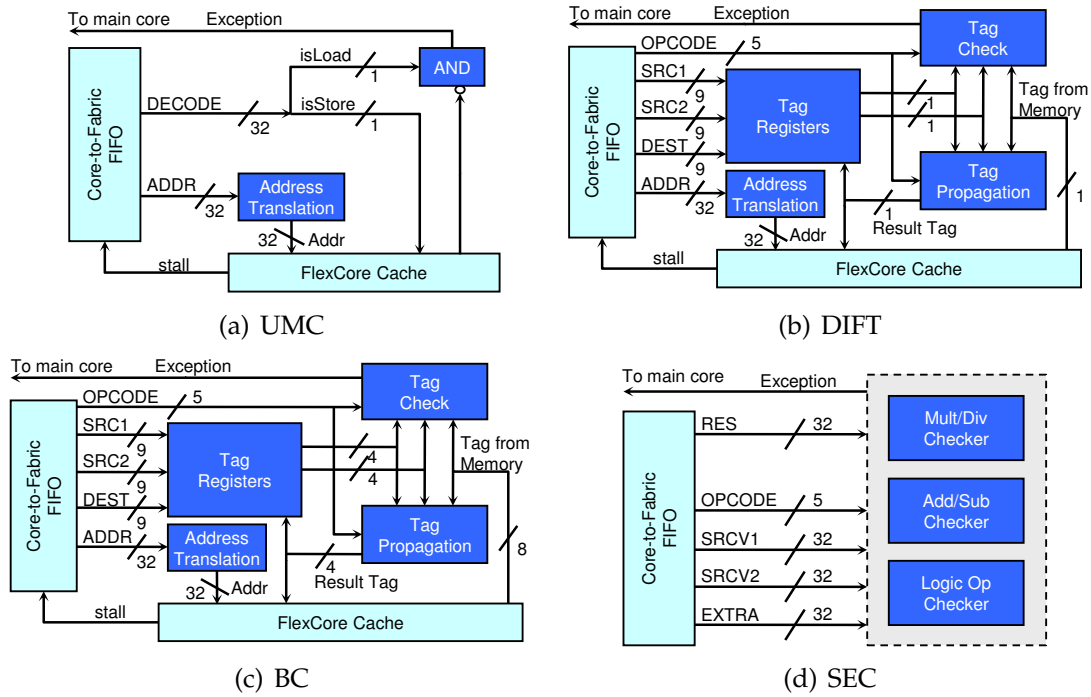
## 3.4 Prototype Implementations

To evaluate the FlexCore architecture, we built a prototype system based on the Leon3 [56] as the main processing core. Leon3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture [128]. The Leon3 architecture provides a single-issue in-order pipeline with seven stages. The core-fabric FIFO interfaces are added to the exception stage, which is the next to last pipeline stage. The prototype does not include a L2 cache, which is not implemented on the Leon3. As this thesis focuses just on the effectiveness of the high-level FlexCore architecture, the current prototype does not support the

meta-data TLB for multi-programming.

The rest of the section describes the implementations of the four runtime monitoring functions that we presented in Section 3.2. The monitoring functions range from simple (UMC) to more sophisticated (DIFT) and area-intensive (BC) and they are moderately pipelined (between 3 to 6 stages) to improve throughput.

### 3.4.1 Uninitialized Memory Check (UMC)




(a) UMC (b) DIFT

(c) BC (d) SEC

Figure 3.4: Block diagrams for FlexCore extension prototypes. Dark blocks represent the FPGA fabric.

Figure 3.4(a) shows the high-level block diagram of the example UMC runtime monitoring function implemented on FlexCore's accelerator fabric. UMC

uses a 1-bit meta-data for each memory word to check if it has been initialized before use [64, 146]. Software explicitly clears the corresponding meta-data on memory de-allocation. The meta-data is updated on a store instruction and checked on a load instruction. For UMC, the core-to-fabric interface is configured to forward only load or store instructions committed by the monitored application to the co-processor. For each FIFO entry received by the co-processor, UMC makes use of the opcode and the memory address accessed by the forwarded instruction. If the opcode indicates a store instruction, the meta-data at the memory destination is updated to 1. If the opcode indicates a load instruction, the meta-data loaded from the metadata cache for the corresponding memory location and is checked. If the meta-data is not set on a load instruction, the check fails, and a trap is delivered to the main processing core through the control interface.

### 3.4.2 Dynamic Information Flow Tracking (DIFT)

Figure 3.4(b) shows the high-level block diagram of the example DIFT runtime monitoring function implemented on FlexCore's accelerator fabric. This prototype for information flow tracking maintains a 1-bit metadata per word of memory and register, which is sufficient to detect control pointer attacks [131, 41][2]. Software can explicitly set or clear metadata, such as when a system call is performed to read input from an untrusted I/O channel. DIFT makes use of the register file on the accelerator fabric to store meta-data for each register on the main processing core. DIFT can also be configured with control registers to spec-

[2]DIFT implementations may use multiple bits per tag [43], or have a tag per each byte in memory [26] for greater detection capabilities. However, the basic operations are almost identical and this discussion applies to those variants in the same way

ify custom metadata propagating and checking policies. For DIFT, the core-to-fabric interface is configured to forward loads, stores, ALU instructions, indirect jumps, and other custom instructions from the monitored application to the co-processor. For each FIFO entry received by the co-processor, DIFT makes use of the opcode, register operand indexes if the instruction read from or stored to the register file, and the memory addresses if the instruction read from or stored to memory. For unary ALU, load, and store instructions, the meta-data is copied from the source metadata register or memory location to the destination meta-data register of memory location. For ALU instructions with multiple source operands, the metadata of the sources are OR'd to determine the destination meta-data value. On security critical instructions such as indirect jumps, the co-processor checks the metadata of the register operand used in the instruction, and if the metadata is set, an exception is delivered to the main processing core through the control interface.

### 3.4.3 Array Bound Check (BC)

Figure 3.4(c) shows the high-level block diagram of the example BC runtime monitoring function implemented on FlexCore's accelerator fabric. The heap bounds-checking technique is an adaption of a previously proposed approach that colors pointers and their intended referent objects and detects buffer overflows when a pointer and its dereferenced location do not match in color [33]. BC is implemented in the FlexCore co-processing model in a way similar to UMC and DIFT. In fact, BC can be seen as a combination of UMC and DIFT features. For meta-data, BC makes use of the register file on the accelerator fabric to maintain an 4-bit metadata (pointer color) for each register; for each word in

memory, the co-processor maintains a 8-bit metadata (4-bit pointer color and a 4-bit object color). Software can explicit set the metadata of the created pointer and allocated memory objects on a memory allocation such as `malloc()` for the heap. For BC, the core-to-fabric interface is configured to forward loads, stores, arithmetic instructions (for pointers), and other custom instructions. For each ALU, load, or store instruction received by the co-processor, BC propagates the metadata by either copying the metadata (load/store) or adding or subtracting the metadata of the two source operands (ALU). For load and store instructions, BC loads a 8-bit (two 4-bit) metadata value from memory. The upper 4-bits are propagated to the metadata register file, and the lower 4-bits are checked against the metadata of the register used to compute the memory address. On a load or store instruction, if the meta-data of the register operand does not match the 4-bit (object color) meta-data loaded from memory, a trap is delivered to the main processing core through the control interface.

### 3.4.4 Soft Error Check (SEC)

Figure 3.4(d) shows the high-level block diagram of the example SEC runtime monitoring function implemented on FlexCore's accelerator fabric. SEC verifies computation results from the main core's ALU to detect soft hardware errors such as single event upsets. While verifying other aspects of the execution such as instruction decoding will require more logic, we believe that the ALU checker represents the general characteristics of soft error checks where verification of each instruction is independent. For SEC, the core-to-fabric FIFO is configured to forward all ALU instructions along with their opcodes, source operand values, and results from the monitored application. For each instruc-

tion received by the co-processor, the opcode, the source values, and the result values are used to compute a checksum to verify that the result was computed correctly. The checker verifies each bit individually for additions, subtractions, and logic operations. For multiplication and division, modular arithmetic (mod M), where M is a Mersenne number of 3, is performed to verify that the result was correctly computed [90]. If the result was not verified to be correct, a trap is delivered to the main processing core through the control interface.

## 3.5 Evaluation

This section evaluates the proposed FlexCore architecture using silicon area, power, and performance as metrics. To study the overheads of the reconfigurability, FlexCore is compared to ASIC implementations of each monitor. We do not evaluate the functional effectiveness of each monitor, such as the attack detection capability of DIFT, as that has already been evaluated extensively in previous research [148, 165].

### 3.5.1 Evaluation Methodology:

To estimate the area, power consumption, and operating frequency of the hardware modules for a typical ASIC flow, we used Synopsys Design Compiler (DC) [135] with a 65nm IBM technology design kit. The FIFOs and the SRAM blocks for caches were created by a memory compiler included in the design kit. To estimate the same metrics for monitors implemented on the FPGA fabric, we used Synplify Pro [134] and Xilinx ISE [153]. The tools were used to map each

monitoring function to the Virtex-5 FPGA, which was also manufactured in a 65nm technology, so as to match the technology node that we used in the ASIC flow. Each monitoring function on the FPGA was fully placed and routed, without dedicated components such as the core-fabric interfaces and caches. This synthesis provided the estimates for the operating frequency and the number of LUTs. To compute a rough estimate of the area, we used an estimate of CLB tile area from the model by Kuon and Rose [82]. The model reports that the area of a CLB tile with 10 6-input LUTs in the 65nm technology node is approximately 8,069$\mu m^2$. We used this estimate of 807$\mu m^2$ per LUT and multiplied it by the total number of LUTs used in our design to generate an area estimate. To compute an estimate of the power of the reconfigurable fabric, we used the Virtex-5 Power Spreadsheet [152] using the frequency and number of LUTs from FPGA synthesis. The area and power consumption of the shadow register file are obtained from a memory compiler and included in the estimates for the dedicated FlexCore modules. The power estimates currently use a fixed toggle rate of 0.1 and static probability of 0.5 for both ASIC and FPGA to provide rough comparisons.

To evaluate the impact of each monitor on monitored application performance, we performed RTL simulations of the entire system, including the processing core, caches, FlexCore monitor, memory controller, and off-chip SDRAM. The default configuration includes a Leon3 core with a single-issue 7-stage pipeline, 32-KB L1 instruction and data caches with 32-B lines, a 4-KB meta-data cache with 32-B lines, and a 64-entry core-to-fabric FIFO. The Leon3 caches use a write-through and no-allocate policy. The simulations ran a set of benchmarks from MiBench [60] and other custom kernels that evaluate computational throughout.

### 3.5.2 Area, Power, and Frequency

| | Monitor | Description | Max Freq (MHz) | Area | | Power | |
|---|---|---|---|---|---|---|---|
| | | | | $\mu m^2$ | overhead | mW | overhead |
| Baseline | - | Unmodified Leon3 - 32KB L1 | 465 | 835,525 | - | 365 | - |
| ASIC | UMC | Leon3 with UMC | 463 | 932,118 | 11.6% | 388 | 6.3% |
| | DIFT | Leon3 with DIFT | 456 | 960,558 | 15% | 388 | 6.3% |
| | BC | Leon3 with BC | 456 | 996,894 | 19.3% | 393 | 7.7% |
| | SEC | Leon3 with SEC | 463 | 836,786 | 0.15% | 364 | - |
| FlexCore | Common | Leon3 with FlexCore | 458 | 1,106,967 | 32.5% | 418 | 14.6% |
| | UMC | UMC on Flex fabric (FPGA) | 266 | 90,384 | 10.8% | 21 | 5.8% |
| | DIFT | DIFT on Flex fabric (FPGA) | 256 | 123,471 | 14.8% | 23 | 6.3% |
| | BC | BC on Flex fabric (FPGA) | 229 | 203,364 | 24.3% | 27 | 7.4% |
| | SEC | SEC on Flex fabric (FPGA) | 213 | 390,588 | 46.7% | 36 | 9.9% |

Table 3.3: The area, power, and frequency of the FlexCore architecture. The overheads in silicon area and power consumption are shown relative to the baseline Leon3.

Table 3.3 summarizes the estimated area, power consumption, and operating frequency for the Leon3 processor with and without various monitors. We found that the unmodified Leon3 with 32-KB L1 caches can run up to 465MHz and consume about $0.836mm^2$ and 364.2mW. These numbers are comparable to those of the ARM processors such as ARM926EJ-S [9]. The full ASIC results, where the Leon3 processor with each monitor is synthesized using the ASIC flow, show that UMC, DIFT, and BC consume 12 to 20% additional silicon area and 6 to 8% additional power. These overheads are dominated by the meta-data cache and FIFOs for the core interface. For SEC, the overheads are negligible because SEC does not require a meta-data cache or a complex interface. The Leon3 processor with an FlexCore modification in full ASIC implementations results in a slightly lower operating frequency because of the probes on internal pipeline signals.

For the FlexCore implementations, the table separately shows the estimates for each monitor on the Flex fabric and the estimates for the dedicated (ASIC)

modules common for all FlexCore implementations including the FlexCore interface and 4-KB meta-data cache. Similar to the ASIC implementations, the synthesis results show that the addition of the FlexCore interface that taps into the main core pipeline slows down the frequency slightly by 1.5%. The dedicated FlexCore modules (the interface and the meta-data cache) add about 32.5% more silicon area and 14.6% more dynamic power compared to the baseline Leon3 processor. These overheads are higher than the ASIC implementations because the FlexCore interface is more general. In the FlexCore implementations, the Flex fabric adds noticeable overheads in addition to the meta-data cache and the interface. The monitors require the Flex fabric to be 0.09 to $0.39mm^2$, which represent 11 to 47% additional area overheads, and consume 6 to 10% additional power.

While the relative area overheads are noticeable for the Leon3 processor, which is a tiny embedded processor, the results demonstrate that the FlexCore architecture is far more energy-efficient than running a monitoring function on another processing core. Also, we note that the absolute area and the power consumption for the FlexCore modules are quite small if compared to higher-end microprocessors. For example, each processing core in UltraSPARC T2 occupies $12mm^2$ in the 65nm technology. MIPS R14000, which is fabricated in the $0.13\mu$ technology, occupies $204mm^2$ and consumes 17W at 500MHz. For these modern processors, the relative overheads of FlexCore will be insignificant.

| Benchmark | UMC | | | DIFT | | | BC | | | SEC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | (1X) | (0.5X) | (0.25X) | (1X) | (0.5X) | (0.25X) | (1X) | (0.5X) | (0.25X) | (1X) | (0.5X) | (0.25X) |
| sha | 1.01 | 1.01 | 1.01 | 1.01 | 1.06 | 1.16 | 1.03 | 1.07 | 1.15 | 1.00 | 1.33 | 1.50 |
| gmac | 1.01 | 1.01 | 1.09 | 1.01 | 1.15 | 1.34 | 1.02 | 1.17 | 1.37 | 1.00 | 1.20 | 1.47 |
| stringsearch | 1.03 | 1.05 | 1.12 | 1.16 | 1.46 | 1.89 | 1.22 | 1.45 | 1.84 | 1.00 | 1.00 | 1.11 |
| fft | 1.01 | 1.01 | 1.01 | 1.02 | 1.05 | 1.31 | 1.02 | 1.03 | 1.35 | 1.00 | 1.15 | 1.45 |
| basicmath | 1.01 | 1.01 | 1.01 | 1.03 | 1.08 | 1.34 | 1.04 | 1.07 | 1.37 | 1.00 | 1.14 | 1.43 |
| bitcount | 1.04 | 1.06 | 1.07 | 1.08 | 1.36 | 1.69 | 1.13 | 1.27 | 1.64 | 1.00 | 1.19 | 1.48 |
| geomean | 1.02 | 1.02 | 1.05 | 1.05 | 1.18 | 1.43 | 1.07 | 1.17 | 1.44 | 1.00 | 1.16 | 1.40 |

Table 3.4: The performance overhead comparisons between ASICs and FlexCore. The performance is shown as the execution time that is normalized to the execution time of the baseline Leon3 processor without modifications. The table includes execution times when each monitoring approach is running at the same frequency as the main core (1X), half of the frequency of the main core (0.5X), and one quarter of the frequency of the main core (0.25X).

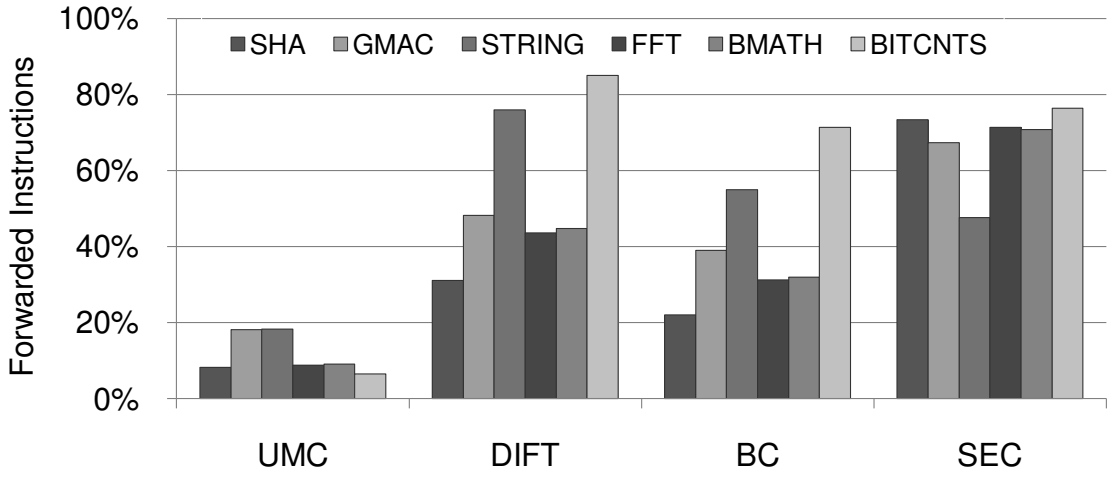



Figure 3.5: The percentage of instructions forwarded to the reconfigurable fabric for each FlexCore monitor prototype.

### 3.5.3 Performance

The performance overheads of the FlexCore implementations are estimated by running RTL simulations with two separate clocks for the main core and the Flex

fabric. The clock frequency for the Flex fabric is set based on the frequency estimates from the synthesis results. BC, UMC, and DIFT run at half the frequency as the main core (0.5X in the table) while SEC runs slower (0.25X). Table 3.4 presents the normalized execution time for each monitor. The execution times are normalized to the baseline Leon3 without any monitoring. The ASIC implementations with the same clock frequency for both monitor and core show at most 7% performance overheads. For UMC, the performance of FlexCore is virtually identical to the ASIC performance despite running at half of the core frequency because only a small portion of instructions are forwarded to the reconfigurable fabric to be processed as shown in Figure 3.5. BC and DIFT have a slightly higher performance overheads of 18% for FlexCore because the fabric needs to process a larger percentage of main core instructions and access the memory for meta-data. SEC has the highest performance overheads at 40% because it processes a large number of instructions while running at a low clock frequency.

The experimental results demonstrate that monitoring on FlexCore is far more efficient than software implementations with comparable capability. For DIFT, previous software implementations have placed the performance overhead as high as 37 times [102]. Even a highly optimized implementation on high-end superscalar processors reported an average slowdown of 3.6 times [113]. For UMC, Purify [64] performs similar checks by adding state bits to each byte in memory, and was reported to slow down monitored programs by a factor of 3. The array bound check in software can also incur a noticeable slowdown in memory intensive programs up to 1.69 times even with extensive optimizations [47]. We also note that these software implementations are tested on high-performance processors where software overheads can be hidden by

the superscalar processing and dynamic scheduling. We expect the software overheads to be even higher for simple in-order processors.

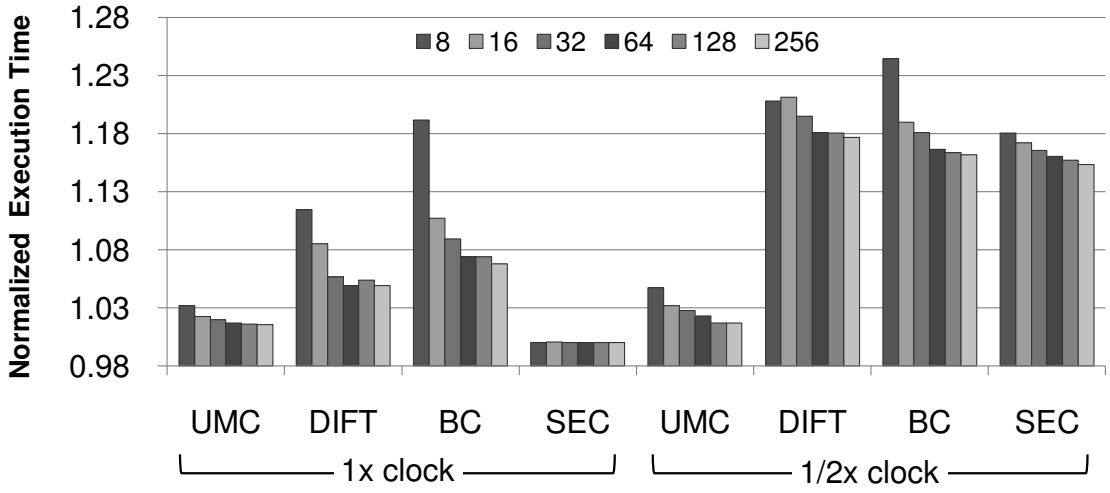



Figure 3.6: Average FlexCore performance for varying forward FIFO sizes.

Since the processor must stall when a forward FIFO between the core and fabric is full, the FIFO must be sized appropriately to accommodate the slowdowns on the reconfigurable fabric for meta-data accesses. For this purpose, we studied the impact of the FIFO size on the performance for our monitors as shown in Figure 3.6. A 64-entry forward FIFO was found to be sufficient for our implementation. FIFO sizes smaller than 64 entries caused noticeably greater performance overheads while larger forward FIFO sizes offered marginal performance benefits. The silicon area for the FIFO only increase by about 10% between the 16-entry FIFO and the 64-entry FIFO because of the SRAM peripheral circuits.

### 3.5.4 Discussion

The simulation results show that the co-processing for security and reliability monitoring is indeed feasible on FlexCore for embedded processing cores running at hundreds of MHz (<500). From the results of the evaluation, we observed that the two main sources of overhead on the monitored application in FlexCore are the throughput mismatches between the main processing core and accelerator fabric, and contention for shared memory bandwidth. The former is apparent in how the performance overheads of each monitor increases when the operating frequency of the accelerator fabric decreases. The latter can be demonstrated by the larger performance overhead of BC, which makes use of larger 8-bit metadata, in comparison with UMC, which makes use of 1-bit metadata. These overheads remain a barrier for the proposed approach's applicability to higher performance processing cores that run at several GHz (or more).

## 3.6 Demo Prototype

The evaluation results of the preceding chapters were gathered from a simulation of the RTL prototype. To flesh out the details of the design and provide for a live demo, we mapped the prototype to a stand-alone FPGA. To meet this goal, we hashed out implementation details such as the fabric-memory interface, exception handling, and extending the ISA with custom instructions. The rest of this section will briefly discuss the prototype's key components, which were implemented in collaboration with Luke Ackerman.

The baseline design used in our prototype was a system made up of a SPARC

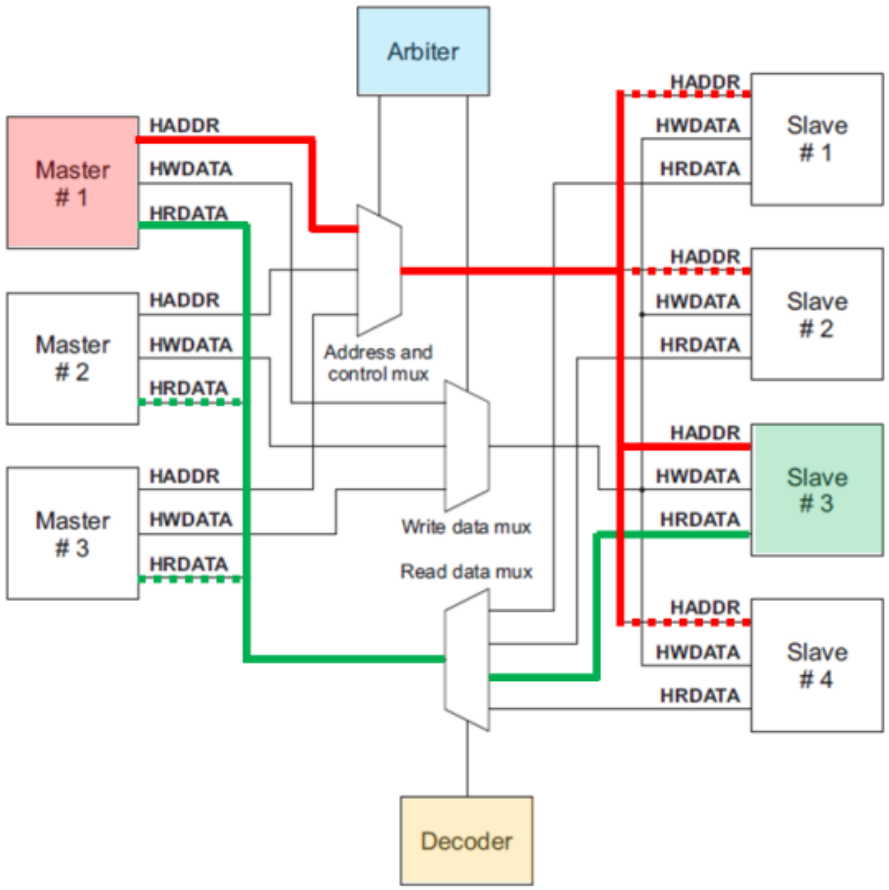



Figure 3.7: AHB usage example: The red lines show how Master #1 is granted control of the bus by the arbiter and broadcasts its request to all AHB slaves. In the example, AHB slave #3 holds the requested data, and the green lines show how it responds to the request and how its data is steered back to Master #1.

processing core that served as the a master on the AMBA High-Performance Bus (AHB) and a range of peripherals, such as memory and PHY controllers, that served as slaves. AHB usage protocols allow a central arbiter to coordinate behavior between AMBA masters and slaves. At a high level, the protocols allow any master to make requests; the arbiter will grant control to one master at a time, the granted master can then broadcast its request to all slaves; when the data becomes available, a decoder will select only the data returned by the designated slave to return to the requesting master. Figure 3.8 shows how the

arbiter and the decoder would handle an example request for data made by an AHB master. To complete an implementation of the prototype, which has a co-processor that can independently access and cache metadata from main memory, required the addition of one more AHB master that conformed to AHB usage protocols and timing.

| Opcode | rs1 | rs2 | rd | Function |
|---|---|---|---|---|
| **CPOP1** | X | X | 0 | Disable Monitor |
| **CPOP1** | X | X | !(0) | Enable Monitor |
| **CPOP2** | A | B | 0 | MEM[A] = B |
| **CPOP2** | A | B | 31 | MEM[A] = INIT (B) |
| **CPOP2** | A | X | 1-30 | RD = TAG[A] |

Figure 3.8: Example usage of special CPOP instructions for monitoring control and metadata initialization

To allow the system to manage runtime monitors implemented on FlexCore, software interfaces must be exposed to allow the system to bootstrap metadata values when the monitored application launches, disable monitoring when the monitored application is interrupted, and to specify custom behavior when the monitored application makes system calls. To implement an example of this functionality, we utilized two existing and unimplemented opcode encodings in the SPARC instruction set, CPOP1 and CPOP2. At a high level, most of these instructions have minimal effect on the pipeline (an exception is CPOP2 when RD is specified). These newly implemented instructions are forwarded to the co-processing fabric, which recognizes them as metadata access requests from the main processing core. Figure 3.8 shows an example implementation of these special instructions for a runtime monitor that performs taint tracking. In this example, CPOP1 is used to enable or disable the monitor based on its RD field, and CPOP2 is used to read or write metadata values.

When errors are detected in the runtime monitor and sent back to the main

core, a mechanism must be available on the main core to catch the error signal and determine the next step to take after the error has occurred. To implement this mechanism, we extended the trap tables in the SPARC architecture with an additional trap type that is specific to errors that are signaled by runtime monitors. With this functionality, custom software handlers can be registered for invocation on errors reported by the runtime monitor implemented on the co-processor.

After these modifications were implemented, we were able to physically demonstrate the ability of the prototype, which was running on a Xilinx ML509 FPGA, to detect buffer overflow and memory overwrite errors and call the appropriate software handler. Additional details about the steps taken to implement the hardware modifications, timing diagrams, and source code for demo programs are available in Luke Ackerman's project report [2].

## 3.7 Summary

In this chapter, we discussed the design and implementation of FlexCore, a hybrid architecture that combines a custom-built microprocessor with an on-chip reconfigurable fabric for mapping hardware reliability and security functions. FlexCore can serve as a general platform for hardware run-time monitoring features, and these features can be added for testing and/or full-time use long after the platform has been fabricated and shipped. Case studies and prototypes of four example monitoring features on FlexCore show that it can support a broad spectrum of monitoring functions ranging from very simple to more in-depth computational checking. Our performance evaluation with a suite of

monitoring approaches showed that the overheads of FlexCore on monitored application performance correlates with the difference in throughput between the general purpose processing core and the reconfigurable fabric. For simpler hardware runtime monitoring approaches such as information flow and uninitialized memory checking, the difference in throughput is small and FlexCore was shown to be able to have low performance overheads on the monitored application. However, for runtime monitoring approaches that require complex and difficult to pipeline logic, this difference can become significant, leading to higher performance overheads.

CHAPTER 4

# HARDWARE ACCELERATOR FOR HIGH-PERFORMANCE RUN-TIME MONITORING

This chapter proposes an optimized on-chip accelerator architecture for runtime monitoring named Harmoni (Hardware Accelerator for Runtime MONItoring). Unlike FlexCore, which utilizes a bit-level reconfigurable fabric to map any runtime monitoring approach that can be implemented using lookup tables, Harmoni is more purpose-built as a datapath for metadata processing. The narrowed focus of Harmoni allows the accelerator to be used as a co-processor for high-performance processors running at clock frequencies of a few GHz.

The FlexCore architecture presented in Chapter 3 was shown to be a promising approach for security and reliability. In augmenting a general purpose processing core with a bit-level reconfigurable fabric, we reasoned that many runtime monitoring functions perform fine-grained, bit-level operations that can map efficiently to such fabrics. An evaluation of a prototype of FlexCore with several example runtime monitoring functions showed that it was capable of providing functionality to embedded processing cores (< 500MHz). However, despite optimizations to mask the inefficiencies of the reconfigurable fabric, monitoring functions mapped to the FlexCore architecture were unable to attain operating frequencies much higher than 250MHz. The low operating frequencies of the on-chip bit-level reconfigurable fabric limit its applicability to embedded applications.

From a high-level perspective, there appears to be a fundamental tradeoff between efficiency and programmability for runtime monitoring as illustrated in Figure 4.1. Programmability requires additional hardware (multiplex-

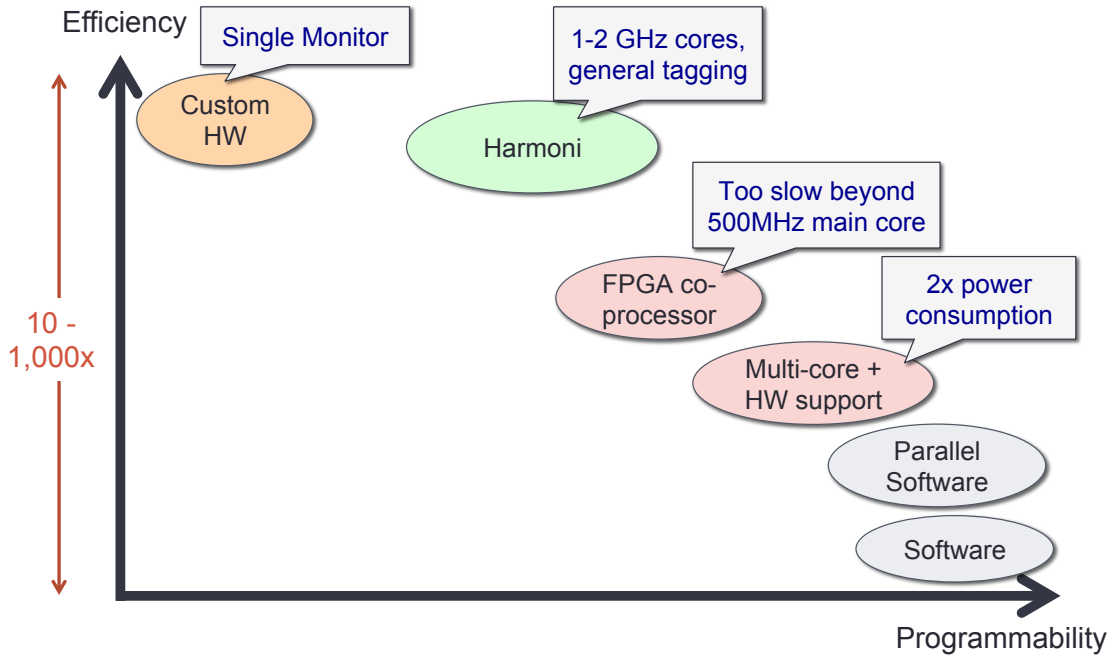



Figure 4.1: Trade-off between efficiency and programmability.

ers, switches, routing, and so on) so as to allow for choices; but the additional hardware (for example as on a programmable gate array) results in lower operating speeds compared to equivalent ASIC designs for implementing complex logic. However, the performance gap between island-style bit-level reconfigurable fabrics and ASICs is large, and there appears to be an opportunity to bridge the gap by trading off some flexibility and choice for performance.

To this end, we propose to carefully restrict the programmability of the accelerator fabric so as to better match the characteristics of common runtime monitoring operations. In particular, we found that many instruction-grained monitoring techniques are built upon the notion of tagging, which binds metadata to program state. Using these bindings, runtime monitoring can track and check the state of the computation by updating and checking the metadata based on

the events that occur on relevant program state in the monitored application. While the notion of tagging has been studied before [58, 129, 53, 161], this work presents a more general architecture that can support a broad range of run-time monitoring techniques that are built on tagging.

Harmoni maintains programmability by broadly supporting monitoring schemes that use tagging of various sizes and at different granularities. Harmoni also supports operations on metadata that range from regular ALU computations to irregular updates and checks by using a combination of custom circuits and programmable look-up tables. By focusing specifically on supporting monitoring schemes that make use of tagging, Harmoni can achieve a high operating clock frequency of 1.25GHz on a 65nm standard cell technology. This higher clock frequency allows Harmoni to keep pace with high-performance processors that are running at clock frequencies of a few GHz and maintain low performance overheads. An evaluation of the Harmoni prototype also shows that the architecture is far more energy-efficient compared to using multiple processing cores for both running the application and performing monitoring operations. Harmoni has moderate area overheads for a small embedded processing core, but is quite small compared to modern processing cores that run at higher frequencies.

The rest of the chapter is organized as follows. Section 4.1 describes the model for the parallel monitoring with tagging, and shows how example monitors can map to the model. Section 4.2 presents the tagging architecture. Section 4.3 studies the performance, area, and power consumption of our architecture. Section 4.4 concludes.

## 4.1 Tag-Based Monitoring Model

### 4.1.1 Computational Model

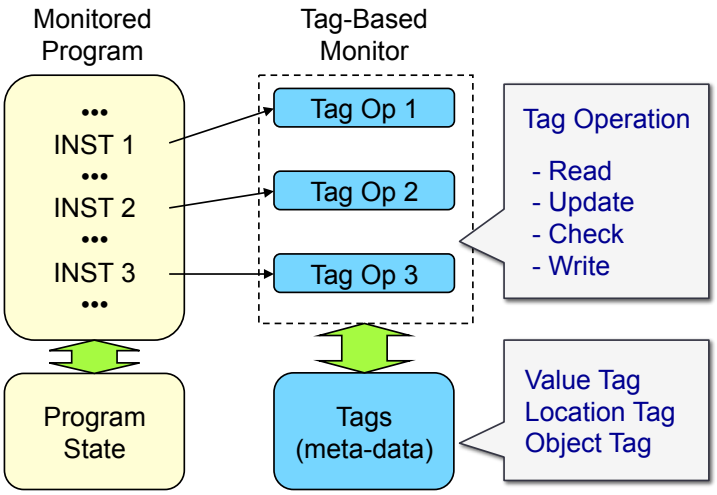



Figure 4.2: Parallel run-time monitoring with tags.

Figure 4.2 shows a high-level model of computation used by instruction-grained monitoring techniques. In the figure, the light blocks on the left represent the computation performed by the monitored application and the dark blocks on the right represent the monitoring function. Conceptually, the monitoring function observes each instruction that is relevant to the monitoring being performed in order to track one or more properties of the monitored application. To track these properties, runtime monitoring approaches can ”tag” or bind individual units of the state of the monitored application to metadata that discretely enumerates the range of the property being tracked. In order to maintain consistency between application data and metadata, runtime monitors must perform bookkeeping operations to access its metadata and to compute new metadata values to reflect the completion of the observed instruction.

In addition to this bookkeeping, runtime monitors must also be able to invoke computation that uses a combination of metadata values and predefined constants to check that any pre-determined invariants of the properties of the monitored application are always being met.

In this section, we will survey several previously proposed monitoring schemes for security and reliability for their bookkeeping requirements and the operations they perform. While not comprehensive of all approaches that have been proposed in prior work, it does cover a spectrum of functionality. From this spectrum we can gather an understanding of the abstract primitives that a generalized accelerator can be built upon.

### 4.1.2 Monitoring Examples

**Dynamic Information flow tracking (DIFT)** [131]: As introduced in Section 3.4.2, DIFT is a security monitoring approach that attempts to prevent software exploits from taking over a vulnerable program by tainting data derived from untrusted I/O channels and tracking their use. To taint untrustworthy data, DIFT binds the value contained within each byte of memory and register with a bit of metadata for each policy that it attempts to use for tracking and checking. In order to track the propagation of taint within the monitored application, various types of policies have been proposed in previous work for computing the metadata of destination operands [131, 29, 34, 43] on arithmetic, logical, move, and branch instructions. On move and arithmetic and logical instructions, the destination operand typically becomes tainted if any of the source operands are tainted. However, this policy alone can be insufficient as

branch instructions can contribute to indirect types of data-flow [29, 34] and false positives are possible when tainted data has been sanity checked or are cleverly cleared using ISA features [131]. Past works have been able to find solutions for false positives by using heuristic knowledge to customize the taint propagation policies and selectively clear the metadata of operands that may have been sanity checked. Lastly, for checking the use of tainted values on sensitive control transfer instructions such as indirect jumps and function calls, the metadata of the source operands are used in logical comparison operations.

**Uninitialized memory checking (UMC)** [146]: As introduced in Section 3.4.1, UMC is monitoring approach based on HeapMon [125], which was proposed as a way to detect programming mistakes where a program variable is not initialized before use [52]. To track whether memory has been initialized since allocation, UMC binds each relevant memory location (of a byte in size) in the monitored application with a bit of metadata to indicate whether the location has been initialized. To compute metadata updates for UMC, the runtime monitor simply sets the metadata of the destination memory bytes on move instructions. On reads of dynamically allocated memory locations, the metadata of the accessed memory bytes are used in logical comparison operations to ensure that they have been initialized.

**Array bounds checking (BC)** [33]: As introduced in Section 3.4.3, BC is an adaption of a previously proposed approach that utilizes a low number of reusable metadata values (instead of a unique metadata value) for each area of memory allocated to detect illegal memory accesses [33]. BC binds a 4-bit metadata to each memory location that encodes the color of the memory location; and a 4-bit metadata to the value contained within each word of memory

and each register that encodes the color of the pointer value contained within the register or memory word. BC relies on the memory allocator to assign the same value (color) to both the metadata of an allocated memory region and the pointer that was initialized using the return value of the allocation function. To accurately track the color of the pointer value resulting from all possible operations on a pointer, BC relies on a complex propagation policy based on the operation performed and the colors of its operands. On memory move instructions, the pointer color is trivially moved from the source to the destination operand. However, on arithmetic and logical instructions, the pointer color of the destination operand is determined by a custom propagation policy. The policy uses heuristics of how pointers and pointer offsets are typically generated to either copy the pointer color or clear the destination pointer color. On all memory access instructions, the pointer color of the address and the memory color of the accessed memory location are used in a logical comparison operation to check that they are equal.

**Reference counting (RC)** [76]: RC transparently performs reference counting in hardware to aid garbage collection mechanisms implemented in software. By transparently maintaining reference counts, RC allows software memory allocation mechanisms to quickly find freed memory blocks and expedite the collection process. RC binds a 32-bit metadata to memory locations that mark the start of allocated memory regions in the memory space of the monitored application. The metadata encodes an integer representing the reference count of the allocated memory region. RC relies on compiler modifications to explicitly indicate instructions that create or overwrite pointer references. On an instruction that creates a new pointer, the pointer value is used to lookup and increment the reference count of the allocation memory region. On an instruction that over-

write an existing pointer, the pointer value is used to look up and decrement the reference count.

| Monitor | Trigger | Action |
|---|---|---|
| DIFT (1-bit value tag) | ALU instructions | Tag(reg_dest) := Tag(reg_src1) or Tag(reg_src2) |
| | LOAD instructions | Tag(reg_dest) := Tag(mem_addr) |
| | STORE instructions | Tag(mem_addr) := Tag(reg_dest) |
| | JUMP instructions | check reg_src1 != "1" |
| UMC (1-bit location tag) | LOAD instructions | check Tag(mem_addr) != "0" |
| | STORE instructions | Tag(mem_addr) := "1" |
| BC (4-bit location tag and 4-bit value tag) | LOAD instructions | check Tag(mem_addr) == Tag(reg_src1) |
| | | Tag(reg_src1) := Tag(mem_addr) |
| | STORE instructions | check Tag(mem_addr) == Tag(reg_src1) |
| | | Tag(mem_addr) := Tag(mem_src1) |
| | ADD instructions | Tag(reg_dest) := Tag(reg_src1) + Tag(reg_src2) |
| | SUB instructions | Tag(reg_dest) := Tag(reg_src1) - Tag(reg_src2) |
| | OR instructions | Tag(reg_dest) := 0 |
| | XOR instructions | Tag(reg_dest) := 0 |
| | NOT instructions | Tag(reg_dest) := -Tag(reg_src1) |
| RC (32-bit object tag) | Create pointer | refcnt[addr] := refcnt[addr]+1 |
| | Destroy pointer | refcnt[addr] := refcnt[addr]-1 |

Table 4.1: Tag types and operations for example set of run-time monitoring functions.

Table 4.1 summarizes the characteristics of each monitoring technique in terms of the metadata bindings and the computation that they perform using the metadata. From the survey of runtime monitoring approaches, it can be observed that a monitoring approach can be differentiated by its *tag type* and *tag operations*. The tag type defines the metadata and metadata bindings that are maintained by the monitor. The set of tag operations define which events in the monitored application trigger compute operations in the monitor and how metadata are updated and/or checked on such events.

### 4.1.3 Tag (Meta-data) Types

To track properties of the monitored application, runtime monitoring approaches can bind each unit of the state of the monitored application to meta-

data that discretely enumerates the range of the property being tracked. In particular, monitoring techniques can maintain metadata for three types of application state: data value, memory location, and high-level program objects.

**Value tag**: Many monitoring techniques tag each data or pointer value in a program with metadata. For example, while DIFT tags each byte of memory with a 1-bit metadata to indicate whether it is from a potentially malicious I/O channel, fat-pointer-based array bounds checking approaches can tag each pointer word with 4 or more bits of metadata to pair it with its intended referent. For both these approaches, they can be viewed as maintaining *value tag*s for individual units of register and memory state in the monitored application, as the value tag follows the value as it is used or copied.

**Location tag**: A monitoring approach may also tag a location such as a memory address with metadata instead of tagging the value. Such a *location tag* is often used to maintain information on the properties of the storage location. For example, a memory protection technique can bind permission bits to individual memory locations and use these permission bits to check if an access can be allowed regardless of the value contained within the location. Monitoring functions that check for memory usage errors may also use a location tag to check if each memory location is initialized before a read. Similar to the value tags, the location tags are generally bound at a granularity consistent with the smallest unit of data accessed and used by the monitored application.

**Object tag**: For efficiency reasons, program monitoring schemes may choose to not bind metadata to all data of the monitored application, but instead at a coarse-grained granularity for only high-level programming constructs such as classes, structures, arrays, and so on. As an example from the survey, a refer-

ence counter for garbage collection can be maintained for program objects, and metadata representing the counter value can be bound to the address of its first element. While it is possible to implement such coarse-grained tags using fine-grained per-byte or per-word tags–essentially, make all tags that correspond to a large object to be the same value–managing and updating the *object tag* separately reduces the memory space overheads of metadata and reduces the complexity of managing metadata consistency.

### 4.1.4 Tag Operations

In general, *tag operations*, which refer to computation that is invoked to update or check metadata values, are triggered by computation performed by the monitored application that use or update the values and/or locations within the monitored application. Information about the values or locations used in a program can be deduced from the operands of the instruction that executes in the monitored application. For example, ALU instructions indicate that bookkeeping operations need to be invoked to update the metadata of the destination value or location that the computed result was written to. The invoked bookkeeping operations can use the metadata bound to each of the instruction's operands to determine the updated metadata value. Alternately, load/store instructions indicate that a particular memory location is being accessed and can trigger metadata operations that can check the legality of the access using the metadata bound to the memory address and the metadata of the operands used to calculate the memory address. Hence, for each instruction executed by the monitored application that is relevant to the monitoring function being performed, the tagging operations typically consists of a subset of the the following

common sequence of operations.

- **Read**: The monitor reads metadata that corresponds to the value or location used by the monitored application: register or memory for value tags, memory for location tags, and a separate data structure for object tags.
- **Update**: The monitor updates metadata based on the event that occurred in monitored application.
- **Check**: The monitor may check the metadata for an invariant and assert a signal if the invariant is violated.
- **Write-back**: The monitor writes back the updated metadata. The value tag is typically written to the metadata that corresponds to the result of the monitored application's instruction. The location tag is often written to the location that is accessed by the monitored application.

## 4.2 Architecture Design

### 4.2.1 Overview

In this section, we present the design of Harmoni, which is designed as a accelerator for efficiently implementing runtime monitoring techniques based on the notion of tagging. As shown in Figure 3.2, Harmoni is designed to be used as a replacement for the bit-level reconfigurable fabric used in the FlexCore architecture proposed in Chapter 3. The architecture allows the monitoring on Harmoni to be decoupled from the program running on the main processing core using a FIFO within the Core-Fabric interface to buffer forwarded instructions. Using

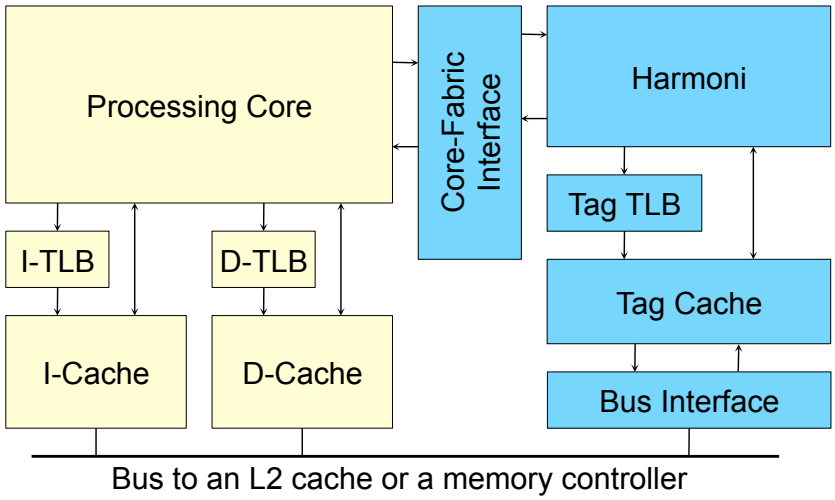



Figure 4.3: High-level block diagram of the Harmoni architecture.

the interface, the main processing core can communicate completed instructions that are relevant to the monitoring being performed in a non-blocking fashion. As long as the FIFO within the Core-Fabric interface is not full (and checking operations events that require synchronization), the monitored application and monitoring on Harmoni can operate independently. In our implementation of the Harmoni prototype, the FIFO is sufficiently sized (64 entries) to accommodate short-term differences in the throughput between the main processing core and Harmoni.

### 4.2.2 Programmability

Figure 4.4 show the block diagram with major components in the Harmoni pipeline. At a high level, the Harmoni is built upon primitive components with knobs for customizing the behavior of each component. These knobs can be finely adjusted so as to allow Harmoni to implement a range of tag-based pro-

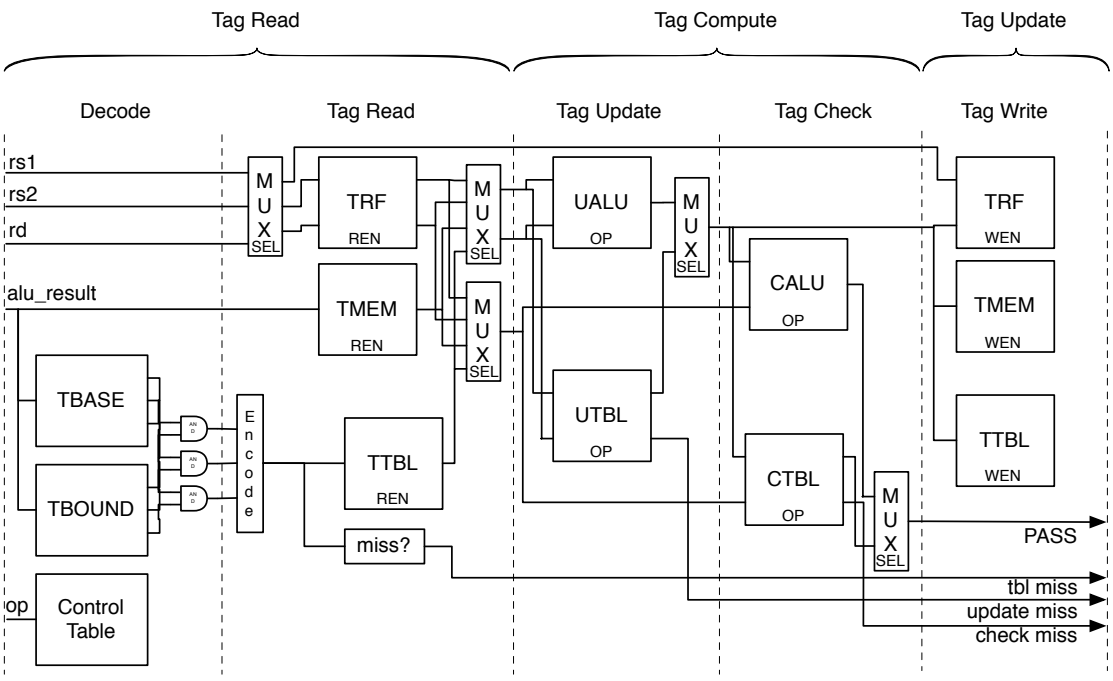



Figure 4.4: High level block diagram of the Harmoni pipeline. The pipeline can be broken down into five discrete stages. The first two stages read the tags of operands used in the instruction, the third and fourth stage update and check the tags, the fifth stage writes the updated tag. The output of the control table is connected to all of the modules in the last four stages of the pipeline and determines their behavior.

gram monitoring operations with different tag types and tag operations.

To efficient support three types of tags: value, location, and object tags, Harmoni allows both the metadata size and tagging granularity to be dynamically reconfigured. The memory and data paths are provisioned for metadata sizes that are a power of two and up to a word (32 bits) in size, check and update computation can be explicitly configured to take only the lower bits that are relevant to the monitoring being performed. The metadata memory translation mechanism can make use of configuration registers on Harmoni to adjust the gran-

ularity of the metadata bindings. For example, for monitoring approaches that bind one byte of metadata to each double-word (64-bits) of the monitored application, the metadata memory translation mechanism would locate the metadata memory address using a base address for metadata and adding a calculated offset and load one byte from the metadata memory hierarchy at that address. The offset calculation can make use of the configuration register, which allows metadata to be bound to monitored application data from one to eight bytes (power of two), in its calculation and emits the appropriate metadata address. Expanding the size of this register allows still even more coarser-grained bindings to be made for all data in the monitored application.

Separate from value and location tags, object tags require the binding of individual metadata values to a memory range of an arbitrary but bounded size. To implement this functionality, Harmoni stores metadata bound to high-level program objects in object tables (OBJTBL) that are explicitly controlled by the privileged processes that aid the monitoring being performed. Previous studies [96, 161, 137] have shown that arrays and data structures in modern applications have high temporal locality and only a handful of entries in a cache for metadata for these structures are sufficient to minimize the need for software intervention to refill the tables. In the Harmoni implementation, we cache 32-entries in the OBJTBL. Each entry of the OBJTBL stores metadata bound to a tuple that represent the base and bound memory address of a cached high-level object and the metadata for a particular address within an object is looked up in two steps. In the first step, the table compares the base and bound addresses of each entry in the OBJTBL with the address to determine if the a tuple exists in the table where the base address is lower than the address and the bounds address is greater than the address. If at least one entry matches, then the index

of the entry corresponding to the tuple is determined using a priority encoder, and this index is used to access the tag table (TTBL), which holds the metadata values, for the corresponding metadata. On access to the OBJTBL that miss, Harmoni can be configured to either ignore the miss or to raise an exception to indicate to the main processing core that a software handler must be invoked to refill the table.

To support value tags, Harmoni shadows the value stored in each register in the main processing core using a tag register file (TRF) and the value in each memory location using a metadata cache hierarchy (TMEM). The TRF binds metadata to values stored within the corresponding register in the main processing core and is indexed with the same decoded register numbers used by instructions forwarded from the monitored application. Similarly, TMEM is accessed using the same virtual address calculated by the main processing core on load/store instructions, but the virtual address is mapped to metadata memory addresses using the metadata memory translation mechanism [1] and these mappings can be cached in the tag TLB. To minimize contention for cache storage with data from the monitored application and the possibility of polluting cache data, TMEM is also built as a hierarchy of conventional caches and shares only the bus interface to shared main memory with the main processing core's cache hierarchy.

To efficiently support various modes of operation of computation for updating and checking metadata, Harmoni includes hardware provisions for performing two distinct compute operations in a sequence for each instruction forwarded by the main processing core. [2] As an example, on an instruction for-

[1] The metadata memory translation mechanism is built on the same principles as paging in the main processing core's memory hierarchy.

[2] As an alternative for implementing the same functionality, the runtime monitoring frame-

warded from the monitored application, Harmoni can compute the updated metadata for the destination operand and then use the updated operand value to check an invariant of the monitored application.

For compute operations that update metadata values, Harmoni uses either the update ALU (UALU) or a software-configurable update table (UTBL). The UALU is built from an optimized arithmetic unit [69] to handle most 32-bit integer computations using two metadata operands. The UTBL is built as a lookup table for implementing custom computation using the same input operands as the UALU. Similar to the update, the tag check operation can also be performed using one of two compute blocks, the check ALU (CALU) or a check table (CTBL). The CALU supports typical arithmetic operations and the CTBL can be looked up similarly to the UTBL. In our prototype, both UTBL and CTBL are implemented as caches with 32 entries. The check operations also take up to two input metadata values, one of which can come directly from the output of the compute block used to calculate the updated metadata value, but include logic to reduce the output to a single binary signal that can be used to indicate to the main processing core whether the check succeeded or failed. This signal is used to deliver an exception over a backward FIFO to the main processing core, which invokes an appropriate exception handler to check for false positives and/or handle the error.

To support the co-existence of value and location tags, Harmoni must also support monitoring functions that read and write TMEM for the same instruction. For example, on memory writes in BC, the legitimacy of the write access

work can be configured to determine the necessary monitoring operations within the clock domain of the main processing core and then forward multiple operations in sequence to Harmoni for the same functionality. However, this alternative can exacerbate the difference in throughput between the main processing core and the monitoring being performed and undermine one of the primary goals of the approach.

must be verified before allowing the write to complete. After the write completes, the metadata of the corresponding address must also be updated. Harmoni provisions for this scenario by allowing TMEM to be read and then used in computation for checking and updating the metadata value. The final computed metadata value is then written to TMEM in another pipeline stage. To get around the structural hazard this functionality represents, we used a multi-port implementation of the first level caches in Harmoni, but this may be unnecessary if separate cache read and cache write queues are used in the TMEM implementation.

To tie together the operations of the pipeline, Harmoni uses a statically-programmed look-up table for pipeline control signals (CONTBL). The CONTBL is indexed by the opcode of the forwarded instruction and its output determines the control signals in each pipeline stage of Harmoni (Our prototype separated the SPARC ISA into 32 instruction types). The control signals from the CONTBL are propagated each pipeline stage to enable or disable operations to read the metadata value, update and/or check the metadata, and whether to stored the computed metadata value. As an example, for an instruction forwarded by the main processing core that will update the metadata of the corresponding destination operand, the CONTBL signals specify whether the computation will be handled by the UALU or the UTBL and what compute operation will be performed.

### 4.2.3 Tag Processing Pipeline

Having described the Harmoni architecture at a high-level, we now describe individual pipeline stages of the Harmoni architecture, which is also shown in Figure 4.4. The Harmoni pipeline can be broken down into five high-level stages. The first two stages read the relevant metadata for the monitored instruction, the third stage computes the updated metadata, the fourth stage performs a metadata check, and the fifth stage writes the updated metadata value back to the tag register file, the tag memory, or the object tag table. In contrast to typical processor pipelines that begin with fetch and decode stages, Harmoni obviates the need for much of these stages by following the control flow of the monitored application.

In the first stage of the pipeline, the instruction is "decoded". The CONTBL is accessed using the opcode of the forwarded instruction. The tag register file indexes to read tags from are selected using signals from the control table. At the same time, the stage looks up the OBJTBL by checking the base and bound addresses with the pointer address from the main processing core.

In the second stage of the pipeline, metadata values are accessed from the tag register file (TRF), the tag memory (TMEM), and the object table (OBJTBL). Up to two metadata values are read from the TRF, one contiguous chunk of metadata values are read from TMEM, and one object metadata value is read from the OBJTBL.

In the third stage of the pipeline, the computation to update the metadata value is performed. Up to two metadata values are used by either the UALU or the UTBL to calculate the updated metadata value. The output of either the

UALU or the UTBL is selected using control signals from the CONTBL and propagated to the next stage of the Harmoni pipeline.

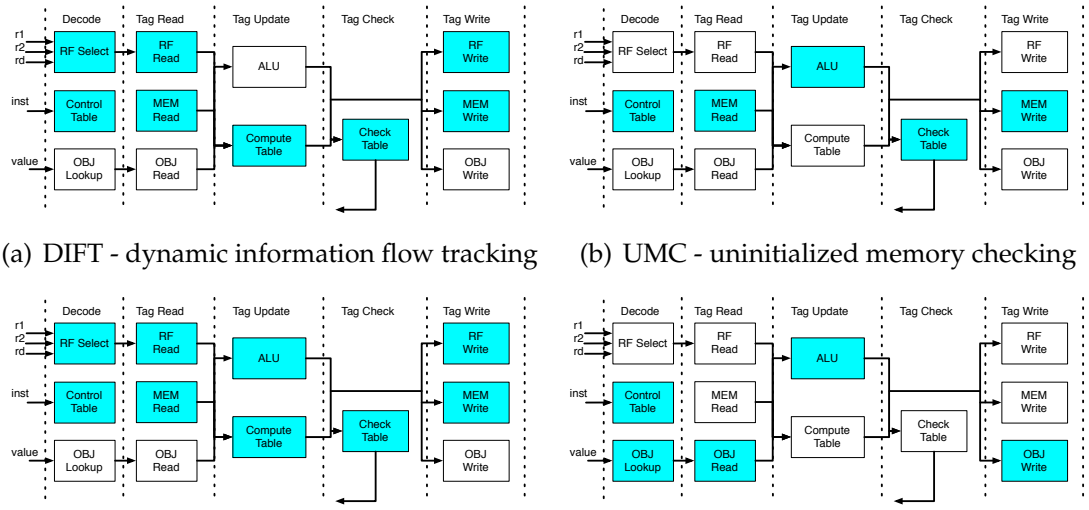
In the fourth stage of the pipeline, the metadata value is checked. The CALU takes the updated metadata value along with another metadata value from the TRF, the TMEM, or the OBJTBL, and performs a unary or binary comparison to determine the check result. The CTBL uses the same input tags to perform a more customized checking operation. The output of either the CALU or the CTBL is selected using control signals from the CONTBL to drive exceptional signals back to the main processing core.

In the fifth and final stage of the pipeline, the updated metadata value is written back to the tag register, the tag memory, and/or the object tag table. The updated metadata is sent on a broadcast bus to these three structures, and the writing of this metadata for each module is controlled by a set of control signals generated from the CONTBL.

### 4.2.4 Monitor Examples

Figure 4.5 shows how the run-time monitoring techniques that we discussed in Section 4.1 can be mapped to Harmoni. The figure highlights the modules that are used by each monitoring scheme in block diagrams.

The modules that are used by uninitialized memory checking (UMC) are shown in Figure 4.5(b). In UMC, the location tag of the memory that is accessed is read and checked on a load, and the location tag of the accessed memory address is set on a store. For load instructions, the CONTBL enables reading of




(a) DIFT - dynamic information flow tracking

(b) UMC - uninitialized memory checking

(c) BC - memory bounds checking

(d) RC - hardware support for reference counting

Figure 4.5: Run-time monitoring techniques mapped to the Harmoni co-processor.

the tag of the accessed memory location from the TMEM. This tag is propagated through the UALU unchanged, and checked in the CALU to confirm that the accessed memory location was initialized (the tag is set). For store instructions, the control table sets the UALU to output a constant ”1”, which is stored to the TMEM at the address from the store.

The modules used by dynamic information flow tracking (DIFT) are shown in Figure 4.5(a). In DIFT, ALU instructions propagate metadata between registers, memory instructions propagate metadata between registers and memory, and metadata is checked on certain sensitive control transfer instructions. For ALU instructions, the CONTBL selects decoded register file indexes sent from the monitored application and metadata values read from the TRF are sent to the UALU. The UALU is programmed to compute the updated metadata based on the metadata of the inputs and the instruction opcode by performing an OR

operation, and the result is written back to the TRF. For load instructions, the CONTBL enables reading of metadata from the TMEM and sending the metadata from memory to the UALU. The UALU passes through the metadata unaltered and this result is written to the TRF using the destination register index for the load. For store instructions, the CONTBL enables reading of the metadata from the TRF. This metadata is propagated through the UALU and into the TMEM. For indirect jump instructions, the metadata of the jump target address is read from the TRF, propagated through the UALU, and checked in the CALU.

The modules used by array bounds checking (BC) are shown in Figure 4.5(c). In bounds checking, explicit instructions set and clear a value tag (pointer tag) and location tags (corresponding locations) on memory allocation and deallocation events, the pointer tags are propagated on an ALU instruction and a load/store operation, and then the pointer and location tags are compared on each memory access instruction to ensure in-bound accesses. In our prototype, we implemented the scheme using 4-bit metadata values, which represent 16 colors. However, the necessity of needing to support value and location tags from BC meant that 8-bit metadata values need to be maintained for each word of memory where the 4 MSB represent the location tag and 4 LSB represent the value tag. For ALU instructions, the value tags (pointer colors) of source operands are read from the TRF and propagated to the UALU. The UALU calculates the tag for the result, and this tag is written to the TRF for destination register. For memory load instructions, the CONTBL enables both the TRF and the TMEM in the second state to read both the value tag of the load address (TRF) and the value and location tags of the accessed memory location (TMEM). Then, the pointer tag of the memory address is compared with the memory location tag from the TMEM in the CALU to ensure that they match. The tag of

the loaded memory value is then written back to the TRF. For memory store instructions, the pointer tag of the accessed address is read from the TRF and compared with the memory location tag from the TMEM as in the load case. The tag of the value that is being stored is then stored to the TMEM. To support customized metadata propagation policies that are necessary to avoid false positives [33], the UTBL can selectively leveraged to support the exceptional cases.

The modules used by hardware reference counting (RC) are shown in Figure 4.5(d). In the reference counting, specialized instructions that create or overwrite a pointer explicitly send the pointer value that was created or overwritten to the monitor running on Harmoni. The pointer is compared to a stored list of object base and bound addresses in the OBJTBL to determine the reference count (metadata) that needs to be updated. If the pointer does not lie within the base and bound addresses of any objects in the OBJTBL, an exception is raised so that software on the main processing core can update the OBJTBL. For instructions that create a pointer, the object that the created pointer points to is looked up and the reference count for that object is incremented in the UALU. This updated reference count is written back to the OBJTBL. For instructions that overwrite a pointer, the object that the overwritten pointer points to is looked up, the reference count for that object is decremented in the UALU, and this updated reference count is written back to the OBJTBL. The reference counts can be utilized by the garbage collection approach running on the main processing core in two ways: the collector can send custom instructions to the monitor to query is a particularly object can be collected; or an exception can be delivered to the main processing core when the reference count for an object has been decremented to zero so that it can be marked for collection.

## 4.3 Evaluation

| **Leon3 Processor** | |
|---|---|
| Pipeline | 7-stage, in-order |
| Instruction cache | 32 KB, 4-way set-associative |
| Data cache | 32 KB, 4-way set-associative |
| Cache block Size | 32 B |
| Cache write policy | write-through |
| Register file | 144 registers, 8 windows |
| **Harmoni Pipeline** | |
| Control table | 32 entries (28 bits per entry) |
| UTBL | 32 entries |
| CTBL | 32 entries |
| **Harmoni Support Structures** | |
| Core-Harmoni FIFO | 64 entries |
| Tag cache | 4KB, direct-mapped |
| Tag cache block size | 32B |
| Tag cache write policy | write-back |

Table 4.2: Architecture parameters.

| Description | Max Freq (MHz) | Area | | Power | |
|---|---|---|---|---|---|
| | | $\mu m^2$ | overhead | mW | overhead |
| Unmodified Leon3 - 32KB IL1/DL1 | 465 | 835,525 | - | 365 | - |
| Harmoni: Pipeline | 465 | 248,818 | 29.9% | 49 | 13.4% |
| | 1250 | 268,995 | 32.2% | 124 | 34% |
| Harmoni: FIFO, cache, etc. | 458 | 271,442 | 32.5% | 53 | 14.6% |

Table 4.3: The area, power, and frequency of the Harmoni architecture with different maximum tag sizes. The overheads in silicon area and power consumption are shown relative to the baseline Leon3 processor.

To further evaluate the Harmoni architecture, we implemented a prototype system based on the Leon3 microprocessor [56]. Leon3 is a synthesizable RTL model of a 32-bit processor compliant with the SPARC instruction set [128]. The main processing core in Leon3 is composed of an in-order, single-issue seven stage instruction pipeline with 32KB of on-chip L1 instruction and data caches. In the prototype system, completed instructions are forwarded from the exception stage of the Leon3 processing core to the runtime monitor on Harmoni. Because the opcode in the SPARC ISA can come from different parts of the encoded instruction and are irregular in size, we extended the decode logic of the Leon3 processor pipeline to drive a five-bit signal that categorizes instructions

in the SPARC ISA into 32 custom categories. This five-bit signal is used along with a configuration register to ensure that only dynamic instructions from categories that are relevant to the monitoring function being performed are forwarded from the Leon3 processor to Harmoni. To evaluate the area, power, and maximum frequency of this architecture, we synthesized Leon3, Harmoni, and corresponding hardware support structures in Synopsys Design Compiler using IBM/Virage 65nm standard cell libraries. Table 4.2 summarizes the parameters that we used in the evaluation. The power estimates are collected from using a fixed toggle rate of 0.1 and static probability of 0.5 to provide rough comparisons. Table 4.3 shows the results of this analysis.

Even without extensive optimizations, the Harmoni pipeline can run a maximum frequency of 1.25 GHz, which is more than 2.5 times the maximum frequency of the Leon3 processing core for the same synthesis options and standard cell library. This result shows that Harmoni can keep pace with processing cores that have much higher operating frequencies and instruction throughput. Support structure for the Harmoni architecture, including the FIFO interface from the main core and a tag memory system, was synthesized with the Leon3 processing core and had a minimal impact on the core's clock frequency.

Part of the tradeoff for higher throughput is that the Harmoni architecture has noticeable area and power overheads in relation to the baseline Leon3 processor. The total area that includes the core-fabric FIFO, the tag cache, and Harmoni makes up an additional 55% in area compared to the baseline Leon3 processor [3]. The area overhead can be mitigated by further trading off some flexibility and limiting the maximum size of the tags that Harmoni can support. By going to 16-bit and 8-bit pipelines for tag updates and tag checks, the respective

[3] Although the Leon3 is a small and simple embedded processing core

area overheads of Harmoni can be reduced to 44% and 39%. The performance of the Harmoni architecture allows it to be easily coupled with much larger and state of the art processing cores that typically run at a few GHz. For example, the Intel Atom processing core [35] is more than 25 times larger than Leon3 while running at just 2X the maximum frequency of Harmoni. The full 32-bit Harmoni pipeline would present an area overhead of less than 3% for Atom. Moreover, we note that the Harmoni architecture is far more energy efficient compared to an approach that utilizes a regular processing core as a monitor. At 465MHz, Harmoni is estimated to consume 46mW, which is less than 15% of the baseline processor power consumption. This is more efficient than consuming twice the power using two identical cores for both computation and monitoring.

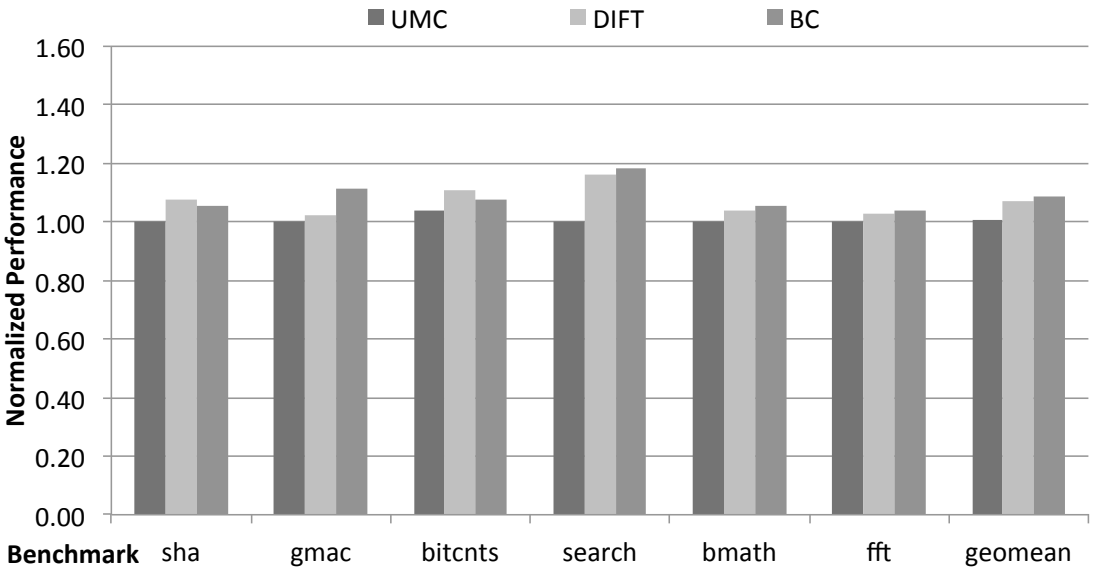



Figure 4.6: The performance overheads of run-time monitoring on the Harmoni co-processor. The Y-axis shows normalized performance relative to an unmodified Leon3 processor. The X-axis shows the names of benchmarks used in the evaluation.

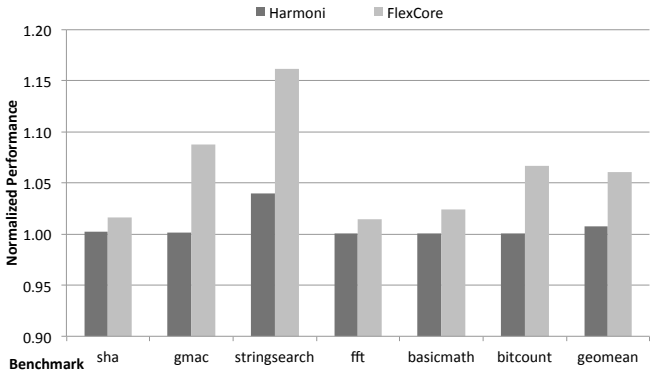
To evaluate the performance overheads of the Harmoni architecture, we performed RTL simulations of the architecture with three different monitoring tech-

niques. Benchmarks used in the RTL simulations include programs from the MiBench [60] benchmark suite as well as two kernel benchmarks for SHA-256 and GMAC, which are popular cryptographic standards. We compared the execution time of these benchmarks between an unmodified Leon3 processor and the Harmoni architecture, Leon3 with hardware monitors mapped to Harmoni, and Leon3 with a hardware monitor mapped to the FPGA fabric as in FlexCore 3.

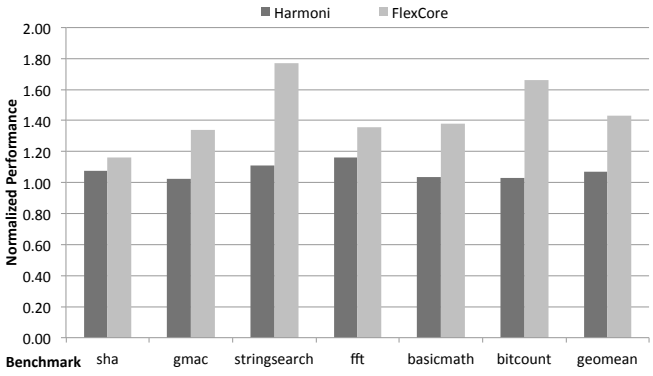
Figure 4.6 shows the normalized execution time of benchmarks on Harmoni with respect to an unmodified Leon3 processing core. We implemented three monitoring techniques on Harmoni, including uninitialized memory checking (UMC), dynamic information flow tracking (DIFT), and array bounds checking (BC). The results show that run-time monitoring on Harmoni has low performance overheads on the monitored program. In fact, the Harmoni performance is virtually identical to that of custom hardware monitors because most overheads come from tag accesses to memory, which are necessary for both implementations.

Because the Harmoni architecture is capable of running at a high clock frequency (as shown in Table 4.3), it is more capable of keeping pace with faster processing cores compared to FlexCore. Figure 4.7 shows the normalized performance of Harmoni on a main processing core with a high clock frequency (1GHz) and compares the result with the FlexCore approach with an on-chip FPGA fabric, which can only run at roughly one-fourth of the main core's clock frequency. Due to its low clock frequency, FlexCore monitoring introduces significant performance overheads. Because Harmoni can match the main processing core's clock frequency, its performance impact is fairly low. For the major-

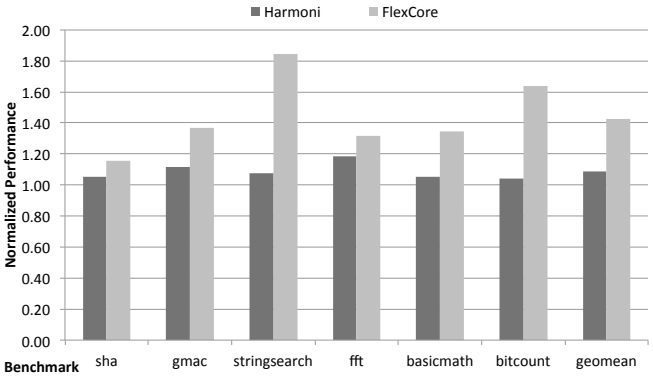
ity of benchmarks that we ran in the evaluation, the performance overheads of Harmoni were much lower than FlexCore with a slow on-chip FPGA. While not shown here, we found that Harmoni can keep pace even with a main processing core running at 2.5GHz, which is twice its maximum frequency, with low overheads. The higher frequencies achievable by Harmoni allows it to be used with high performance processing cores running at a Gigahertz and more.




(a) UMC




(b) DIFT




(c) BC

Figure 4.7: Normalized performance overheads of run-time monitoring on the Harmoni co-processor and the FPGA-based co-processor (FlexCore) for a main processing core running at 1GHz.

## 4.4 Summary

This chapter presented the Harmoni architecture, which attempts to improve upon the performance limitations of the FlexCore architecture by pairing the main processing core with an improved co-processing fabric that is purpose-built to implement monitoring techniques based on the notion of tagging. By fixing the organization of the processing pipeline to support just monitoring techniques based on tagging, Harmoni is able to dramatically improve its throughput. However, Harmoni is able to retain flexibility through the use of configurable components, and we showed how a variety of runtime monitoring techniques can be mapped to the Harmoni fabric for checking memory bugs, security attacks, and performing bookkeeping such as the management of system resources. At a high level, Harmoni presents a new design point on the spectrum between performance and flexibility for runtime monitoring approaches; by matching the common characteristics of monitoring approaches based on tagging, Harmoni can achieve very high performance without restricting the capabilities of the monitoring approaches.

We evaluated the overheads of the Harmoni co-processor by building an RTL model and evaluated the application performance with Harmoni monitoring using RTL simulations. Harmoni was shown to be able to run at frequencies of 1.25GHz when synthesized with Virage/IBM 65nm standard cell libraries. This improvement on performance and reduction in overheads comes at a trade-off of noticeable area and power consumption in relation to the baseline Leon3 processing core. However, Harmoni can keep pace with high performance processing cores that are larger and more power hungry, and the overhead will not be as significant in those applications. The RTL performance simulation

results demonstrated for such high-performance cores that the higher operating frequencies of the Harmoni co-processor allows monitoring functions implemented on the co-processor to have low overheads on the performance of the monitored application. Further, the Harmoni performance overheads can be mostly attributed to overheads of memory accesses for metadata, which remains a problem even for hardware monitors built on custom hardware.

CHAPTER 5

# EFFICIENT META-DATA MANAGEMENT

This chapter presents hardware optimizations to improve the performance of runtime monitoring that support large tags. In earlier chapters, we discussed how runtime monitoring can be effective as a general approach for security and reliability. By operating at an instruction granularity, monitoring approaches can observe detailed events in the monitored application regarding the computation, accessed memory, and flow of control. From these events, runtime monitors can check for behavior that deviates from predefined program invariants such as unallocated memory accesses [64], out-of-bounds pointer dereferences [44], and control data corruption [102].

In the the preceding chapters, we described how the adoption of a co-processing fabric that uses reconfigurable hardware can simultaneously address the conflicting requirements of performance and flexibility for runtime monitoring. The reconfigurable fabric provides flexibility to adapt to new threats; performance optimizations such as filtering and decoupled processing allows its performance overheads of the co-processing architecture to almost match that of using custom hardware. However, runtime monitors must also maintain bookkeeping information (metadata) to track properties of the data for the monitored application. For each event in the monitored application that touches data, runtime monitoring approaches must utilize the associated metadata to update bookkeeping or check for correct behavior. Aspects of the metadata, including the mapping granularity and the metadata size, must also be configurable for flexibility. While large metadata provides important functionality, it can also increase the performance overheads of runtime monitoring.

To motivate the need for large metadata, we refer to the **0-1-∞** rule of software design [89], which states that an entity–the metadata–should either be forbidden, allowed to exist as an exception, or support an unlimited number of instances. On the one hand, mechanisms that use a single bit of metadata have been shown to be able to detect software errors such as spatial memory errors [101] or control-hijacking security attacks [131]. However, the limited metadata size also constrained the functionality of these mechanisms. For example, it was shown that no one policy can detect all attacks for information flow tracking mechanisms [78], and depending on the comprehensiveness of data and control flow tracking, too much data can be marked untrusted and generate false positives [131, 34]. Raksha [43] showed how more bits of metadata can address these drawbacks: the additional bits can be used to implement multiple policies that are tracked and enforced in parallel. These parallel policies can reduce both the false positives and false negatives of information flow tracking [104]. Increasing the number of bits further, DataSafe [31] showed how using individual bits to describe whether the associated data can be read, edited, sent in plaintext, and so on, can allow a system with up to ten bits of metadata to provide end-to-end self-protecting data solutions. As another example, undefined value checkers [64, 122] showed how using two bits of metadata for memory safety to track initialization and allocation status allows the overall mechanism to detect spatial and temporary memory safety errors such as double free and dangling pointers. In general, using more bits of metadata can allow runtime monitoring approaches to have more functionality and better error coverage.

Beyond a couple of bits, other types of monitoring approaches have been proposed that inherently rely on support for larger values of metadata. Programming languages with runtime type checking such as LISP have tradition-

ally relied on the use of metadata to encode the dynamic type of each variable in the application [92, 136]. In particular, the LISP machine [92] used metadata in the range of 2-8 bits to encode the type of a corresponding memory word in the application. To retrofit type checking to C applications, a number of works have also proposed using metadata to find and detect memory errors such as those caused by illegal pointer arithmetic errors [98, 75]. For example, Hobbes [22] proposed to detect errors such as buffer overflows by encoding the variable type, which is determined using static analysis and runtime type inference, with a larger number of bits of metadata and checking the type of all operands used in arithmetic operations. However, type checking alone can provide false positives that can be avoided by instead performing the checks when pointers are dereferenced [117, 95, 151]. To accomplish these checks, capability models can use fat pointers [11, 44, 71, 95] that use metadata of a double-word in size to encode the base and bounds addresses of a pointer's intended referent.

Despite the need to support large metadata for flexibility, there is a limit on how many bits can be managed by hardware. For example, to track each byte of memory value that was derived from untrusted input sources, TaintCheck [102] allocates bookkeeping information in the form of a data structure containing the system call number, a snapshot of the stack, and a copy of the data that was written. This large amount of extra state gives metadata the expressiveness to allow TaintCheck to not only detect but also facilitate recovery from a detected error [140]. As another example, Information flow security (IFS) mechanisms [144, 161] are able to provide stronger guarantees of data confidentiality than discretionary policies [149] by categorizing all entities using privilege levels [18] and using those categories to block data movement. However, in the context of systems such as Facebook, with its billions of users and each of

whom would like to restrict the access of profile items to a circle of friends, the data structures representing all of the possible categories can become very large and difficult to manage. To manage the metadata used in these examples, IFS implementations [144, 161] and TaintCheck [102] adopted tracking mechanisms that encode metadata as pointers to data structures that hold the more detailed bookkeeping information. This optimization reduces the storage overheads of metadata by bounding their size to a word or double-word per byte [102] or word [161] of memory. However, this comes at the cost that monitoring operations must invoke software handlers, which can only be circumvented using hardware caching [161, 145, 46].

However, of greater concern is that the flexibility gained from supporting large metadata will exacerbate the performance overheads of runtime monitoring. The primary sources of performance overhead for the co-processing architecture for runtime monitoring presented in Chapter 4 were the long latency of metadata memory accesses and contention for main memory bandwidth with the main processing core, both of which will increase with larger metadata. In this work, we evaluated optimizations on the metadata memory hierarchy of a Harmoni-based system for runtime monitoring to mitigate the overheads of metadata management. To look for optimization opportunities, we analyzed the characteristics of metadata for several runtime monitoring approaches that use large tags and found two distinct characteristics of metadata: **sparsity** and **frequent value locality**. To take advantage of these characteristics, we propose two hardware optimizations for the metadata cache hierarchy. In particular, we designed and evaluated a filtering mechanism named Non-Default Metadata Cache (NDM) and Non-Default Metadata Cache with Eager Read (NDME) that can take advantage of sparsity to filter metadata access and reduce meta-

data memory traffic. Further, we also designed and evaluated a value-based cache compression mechanism named Dynamic Metadata Compression (DMC) that can take advantage of value locality to compress on-chip cache data.. In our evaluation, we found that a combination of the proposed designs–NDME and DMC–are able to reduce the memory bandwidth requirements of runtime monitoring approaches and improve the performance overheads on monitored applications from 86-267% to 3-117%.

The remainder of the chapter is organized as follows: In Section 5.1 we present our observations on the locality characteristics of metadata. In Section 5.2 we present the organization of our cache architecture for efficiently managing metadata in cache and memory. In Section 5.3 we present an evaluation of the overall system in terms of metadata cache miss rates and memory bandwidth requirements. In Section 5.4 we summarize.

## 5.1 Locality Study

### 5.1.1 Baseline Design

Figure 5.1 shows the high-level organization of the hardware runtime monitoring framework, which is organized similarly to the Harmoni architecture proposed in Chapter 4, that was used as the baseline in this chapter. In the framework, instructions committed by the processing core running a monitored application are forwarded to the runtime monitor using a small FIFO in the Core-Fabric interface. The runtime monitor is control-coupled to the monitored application and performs operations for each instruction received from the queue. For operations that require accessing metadata stored in shared memory, the

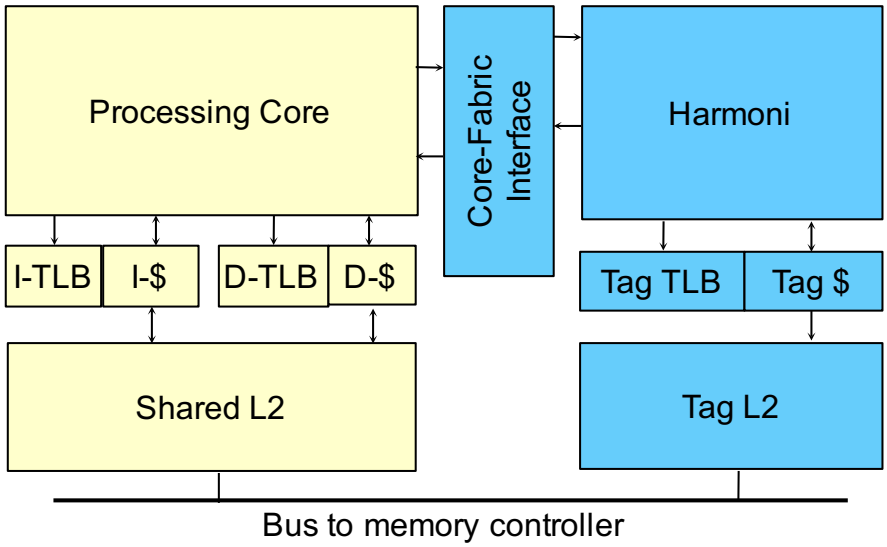



Figure 5.1: Block diagram of the baseline hardware organization for runtime monitoring.

runtime monitor will use a metadata cache hierarchy to access the metadata of the operands used in the instruction. Using a one-to-one mapping between monitored application data and metadata, the runtime monitor can look up the physical memory address of metadata for its operands using a tag TLB. For low latency of access, the tag TLB caches the outcomes of invocations of metadata memory translation mechanisms [100]. The physical address of metadata is then used to access a private tag cache hierarchy that is backed up by shared main memory. The overall organization of decoupled processing and private tag cache hierarchy allows the runtime monitor's operations to be largely decoupled from the main processing core and to mitigate its impact on the behavior of the monitored application.

For efficiency, co-processor in the Harmoni architecture is implemented as an in-order single-issue monitoring pipeline, which research [55] has shown to have sufficient throughput as typical monitoring loads rarely exceeds one in-

struction per-cycle. However, the in-order processing pipeline also means that the effects of misses in the first level cache are more readily exposed as overheads on performance. When misses occur–especially long-latency misses–the instruction queue between the main processing core and Harmoni can fill up and cause the main processing core to stall. In addition, because memory bandwidth is shared with the main processing core for metadata accesses, the monitored application's performance will be affected by increases in memory bandwidth usage for metadata.

To scale the design to processing cores that are capable of scaling to larger applications with larger working sets, it is necessary to augment the design with a secondary cache, which can be scaled up in size without affecting the delay of the first-level cache. While a simple solution may be to increase the size of the L2 cache to match the locality of data, such an approach would introduce high area and power overheads, particularly when metadata is larger in size than the corresponding data. Hence, more efficient management schemes are needed to manage metadata caches. In the remainder of this section, we will analyze the metadata of several runtime monitoring approaches to look for characteristics that can facilitate optimizations for their storage.

### 5.1.2 Runtime Monitors

In this section, we will introduce the example runtime-monitoring approaches that were evaluated in this work. Each of the approaches utilize larger double-word metadata values and manages them in a transparent fashion. To facilitate the description of each of the example runtime monitoring approaches, we will

attempt to distinguish them by the following terminology:

*Metadata Encoding*: the semantic meaning of the metadata for the checks that are being performed. All metadata are zero initially unless explicitly set by privileged software handlers.

*Metadata Propagation*: for a given instruction and the metadata of the instruction's input operands, how the metadata of the instruction's destination operands are updated.

*Metadata Checks*: for a given instruction and the metadata of the instruction's input operands, whether an error should be flagged to interrupt the continued progress of the monitored application.

Because the functionality and the effectiveness of the checks in the example monitoring approaches have been evaluated in previous research, we will focus on evaluating the performance of the monitoring approaches, which are determined by metadata accesses and propagation events.

| Monitor | Threat | Metadata | Propagation | Ref. |
|---|---|---|---|---|
| Flow-source tracking (FST) | tracking untrusted data | {T1, T2} for register and memory bytes | $\{T1_{op1} \perp T1_{op2}$, $T2_{op1} \top T2_{op2}\} \Rightarrow$ Dest (binary) $\{T1_{op}, T2_{op}\} \Rightarrow$ Dest (unary) | [94, 46] |
| Array Bounds Checking (FBC) | spatial memory safety | {UB, LB} for register and memory words | {UB, LB} $\Rightarrow$ Dest | [11, 44] |
| Information Flow Control (IFC) | data confidentiality | Identifier for memory bytes | - | [144, 161] |

Table 5.1: Summary of example monitoring approaches. For FST, binary operations include arithmetic and memory instructions, unary operations include unary arithmetic instructions and move instructions. FBC encodes the intended referent of a pointer as the high address (UB) concatenated with the low address (LB) of the allocated memory chunk.

**Flow source tracking (FST)**

FST is a taint-tracking approach that taints bytes in the monitored application with a metadata value that encodes the source of the data. Unlike information flow tracking, which uses a few bits of metadata [131, 41, 43] and must to resort to fail-stop behavior (and are thus vulnerable to denial-of-service attacks), the metadata managed by FST can enables analysis and recovery techniques [112].

*Metadata Encoding*: FST instruments system calls that read from IO to mark each read from an untrusted input channel with a unique non-zero identifier ($X$). The identifier indicates the time that the input arrived and marks the metadata for each byte of memory derived from the call with a metadata that encodes a range $(X, X)$. This range-encoding increases the size of the metadata to a double-word in size, but allows for transparent metadata propagation when an instruction has multiple source operands that each has non-zero metadata.

*Metadata Propagation*: If the metadata of both of the operands are zero (to indicate trustworthy data), then the destination metadata is also zero. However, if either of the operands of an instruction have non-zero metadata, then the non-zero metadata is propagated to the destination operand. In addition, if both operands of an instruction has non-zero metadata, then the metadata of the destination operand is updated with a range that captures a time span when both inputs were read from untrusted inputs. In other words, for an instruction with operands that have metadata $(X, X)$ and $(Y, Y)$, the destination metadata is updated to $(X, Y)$ if $(X)$ originated from IO that occurred before $(Y)$.

Used in this fashion, the metadata for a register or memory operand used in a failed check can provide information of approximately the inputs from which

the data was derived from. While the encoding is not lossless [94], it does provide a great deal of information about the origins of untrustworthy data.

**Full Bounds Checking (FBC)**

In the C programming language, a memory access error occurs when an access through a pointer goes outside the bounds of the pointer's intended referent. FBC detects such errors using a fat-pointer-based bounds-checking approach [11, 44]. FBC transparently maintains a double-word of metadata for memory words in the monitored application and verifies that accesses using pointers are within the bounds of the pointer's intended referent. Unlike monitoring approaches that can function on a few bits of metadata, capability-based approaches such as bounds checking inherently require a larger number of bits to encode the address range of the intended referent.

*Metadata Encoding*: FBC instruments malloc calls that allocate memory to set the metadata of the pointer returned by the call with the memory range of the allocated memory range. For implicit allocation on the program stack that are more difficult to discern at runtime, we leveraged compiler support [83] that annotate these events to make them explicit. The metadata encoding consists of a double-word that encodes the base and and bounds addresses of the allocated memory chunk.

*Metadata Propagation*: Because the only legal operations on pointers are move instructions and arithmetic with a non-pointer value, the metadata is propagated by copying the metadata of the pointer to the destination.

**Information Flow Control (IFC)**

IFC is based on prior work in information-flow security [144, 161, 32] that protect the confidentiality of user data. Information-flow security mechanisms use metadata to explicitly label all values and storage locations in the monitored application. High-level security policies can be defined by the labels encoded in the metadata, a set of legal flows (pair of labels) that determine permitted flows, and a set of privileges (consisting of labels) held by the monitored application. In order to read from a memory location labeled $(X)$, the process must add $(X)$ to the privileges it holds. However, if the process later attempts to store to $(Y)$, the access is allowed only if the $(X \rightarrow Y)$ can be permitted. In this approach, we assume that the privileges held by the process are managed separately by trusted software. Compared to software-only implementations of information-flow security, hardware support such as IFC can provide benefits such as improved performance, smaller trusted code base, and security enforcement even if the kernel becomes compromised [161].

*Metadata Encoding* To model the functionality of a system protected by IFC, we instrument system calls that allocate memory to set the metadata of bytes in the allocated chunk with a unique identifier. A larger metadata size reduces the possibility of aliasing (and false negatives) that can occur in the set of identifiers used. The metadata encoding used in our evaluation consists of a double-word, which not only addresses aliasing but also allows identifiers to persist across program invocations.

*Metadata Propagation* In contrast to FBC and FST, IFC does not propagate the metadata between data in the monitored application. However, IFC will check the accessed metadata on all memory reads and writes to ensure that the ac-

cess is allowed by the security policy and used to update the process privileges. These checks must be performed by software handlers but have low performance overheads when hardware support is used to cache legal flows [161].

### 5.1.3 Locality Discussion

To determine the characteristics of metadata for the example runtime monitoring approaches, we collected traces of metadata accesses when running a set of C benchmarks in SPEC2000 and SPEC2006 with reference inputs. These benchmark suites were chosen for the evaluation because they are compute and memory-bound and more impacted by runtime monitoring than IO-bound workloads. To keep storage requirements for the traces reasonable, each trace was one and a half billion accesses in length. The following section discusses some of the findings from an analysis of the metadata access patterns and the resulting metadata memory space.

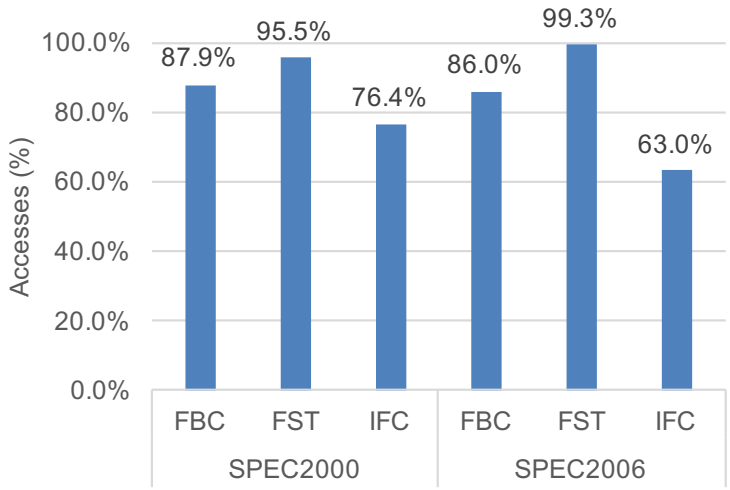



Figure 5.2: The average percentage of metadata accesses for zero for each benchmark suite and runtime monitor.

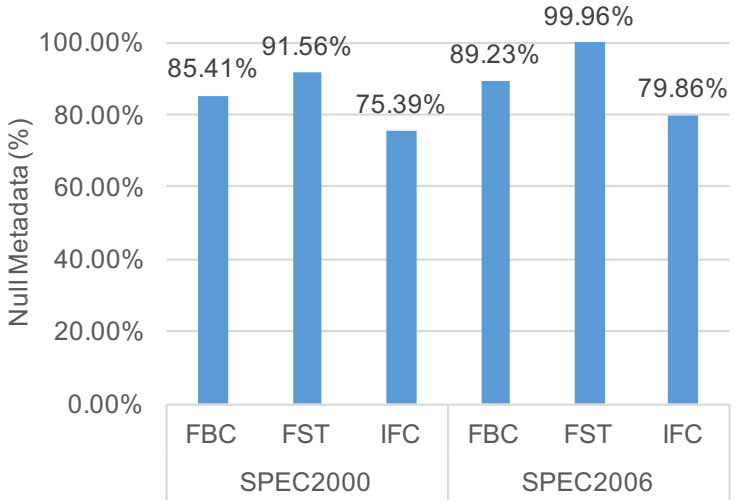



Figure 5.3: The average percentage of metadata memory with zero for each benchmark suite and runtime monitor.

In analyzing the behavior of metadata, we first noticed the abundance of memory accesses for the default value, which is the case where the corresponding memory location in the monitored application is not untrustworthy in FST, not a pointer in FBC, or have default protection in IFC. Figure 5.2 shows that accesses made for metadata with the default values account for a very large percentage of memory accesses made by each runtime monitor. This is further illustrated in Figure 5.3, which shows that a large percentage of the memory space allocated for metadata contain default metadata memory values. These graphs suggest that all three runtime monitors maintain metadata that exhibit sparsity, where much of the metadata remains zero as the monitored application executes; eliminating cache accesses for the default metadata value can circumvent a great deal of memory traffic.

Next, we attempted to observe the temporal locality of metadata accesses by looking at the number of unique metadata values (that are not the default value) observed during the period of execution along with their frequency of

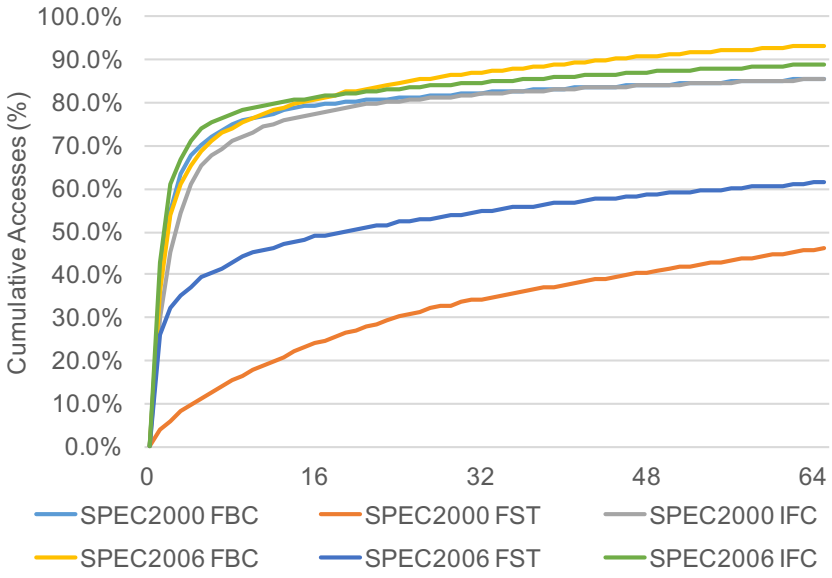



Figure 5.4: Cumulative percentage of metadata accesses for the most frequently observed values - The X-axis shows the number of most frequently observed values; the Y-axis shows the cumulative percentage of metadata accesses where the most frequently observed values are either read from or written to metadata memory.

occurrence. Figure 5.4 shows the cumulative percentage of metadata memory accesses made by the most frequently observed values in the metadata access stream for each runtime monitor and benchmark suite. The results show that the metadata for IFC and FBC exhibit low entropy, where more than 80% of the metadata accesses are to the 25 most frequently observed metadata values. This demonstrates the property of frequent value locality, where a small number of unique metadata values are used for a majority of memory accesses for metadata.

Intuitively, the sparsity and value locality of metadata can be explained by characteristics of hardware support for runtime monitoring, which will perform operations in a transparent manner. This transparency is important for perfor-

mance but can lead to extra metadata accesses for safety: metadata accesses must shadow all application data accesses and will occur even if the metadata had been checked already or is not relevant to the monitoring being performed. (Even if the data is not relevant to the monitoring being performed, returning zero metadata is simpler and more transparent than relying on software intervention [100].) Further, many application operations will change the value of data used by an application but leave the property for which the monitoring approach is checking for unchanged. As an example, an application variable that is used as a loop iterator by a monitored application may have many permutations over the course of execution, but for FBC, the metadata for the variable remains unchanged because the variable is not a pointer.

Lastly, we analyzed the impact of runtime monitoring on the monitored application's performance as the metadata increased in size. For this study, we compared the instruction throughput of the monitored application as the metadata increased for information flow tracking, which has high overheads from performing propagation and checking for almost every instruction. Figure 5.5 shows the results of the study for 1 to 64 bits of metadata for each byte of memory in the monitored application, where 1 bit of metadata matches prior work that marked data only as tainted or untainted [131] and larger metadata sizes reflect the design of FST using 16- and 32-bit values to identify origins. The analysis shows that while the use of 1 bit of metadata results in less than 10% overhead on performance, increasing the metadata size leads to much higher overheads in the monitored application. Furthermore, increasing the size of the second-level metadata caches, which can be scaled up in size without impacting the design of the upper levels, marginally reduces these overheads when scaled up to realistic sizes. Much of the overheads can be attributed to increased num-

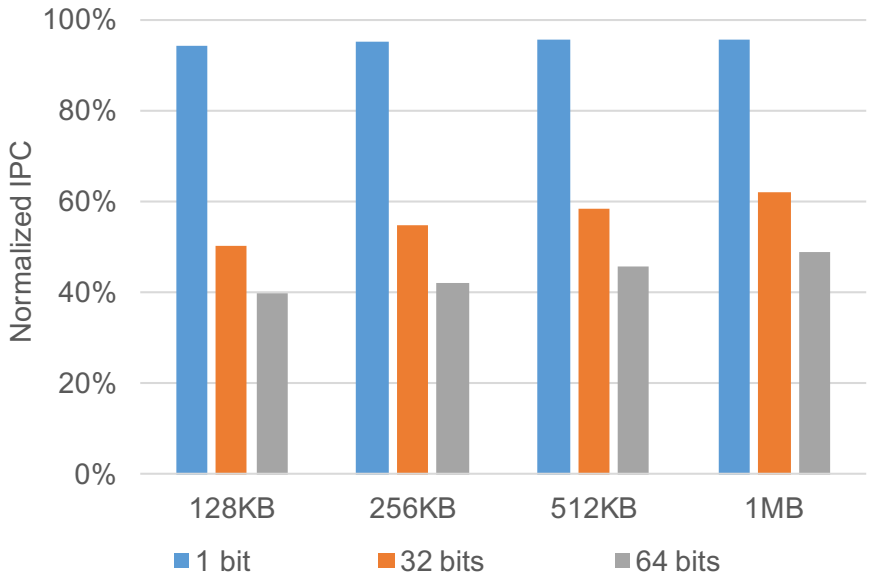



Figure 5.5: Overheads of information flow tracking for increasing metadata sizes normalized to when no monitoring is being performed. The X-axis shows how increasing the size of conventionally-organized second-level metadata caches affects these overheads. Although 1-bit metadata had a low impact, larger metadata sizes resulted in significant overheads on monitored application performance. Increasing the size of the second level cache, which can be scaled up without impacting the design of the monitoring hardware, only slightly reduced these overheads.

ber of metadata accesses that miss to memory and contend with traffic from the main processing core. In particular, using 32-bit and 64-bit metadata led to 24x and 43x average increases in the memory bandwidth usage of FST compared to when 1-bit of metadata is used. In the remainder of this chapter, we evaluate optimizations that can reduce the overheads of large tags without greatly increasing the size of metadata caches.

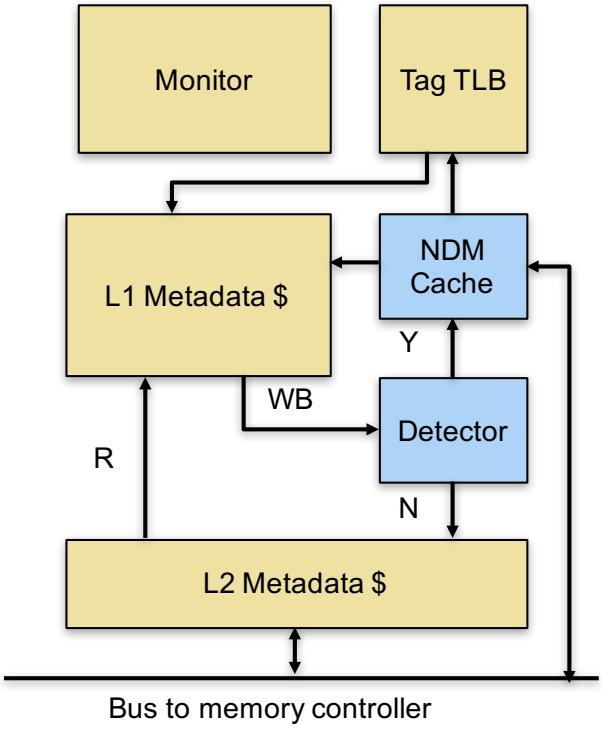



Figure 5.6: Block diagram of the metadata cache hierarchy using the NDM-E cache.

## 5.2 Cache Optimizations

In this section, we will discuss the two approaches we designed and evaluated for optimizing the metadata cache hierarchy. The first approach takes advantage of the spatial locality observed in Section 5.1 to filter out unnecessary metadata accesses. The second approach takes advantage of the frequent value locality in metadata to compress metadata values stored in on-chip caches.

### Filtering Accesses

To take advantage of the sparsity observed in Section 5.1, we propose to add 1-bit of metadata for each contiguous chunk of the metadata memory space that is

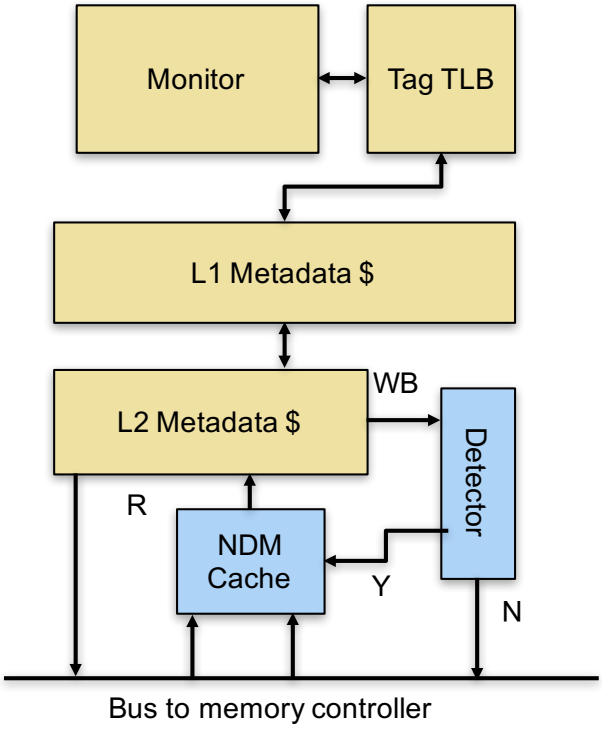



Figure 5.7: Block diagram of the metadata cache hierarchy using an NDM cache.

aligned to the size of a cache line (64-bytes) and use this bit to filter out metadata accesses. The benefit of using a bit for a memory chunk of a cache line in size is that it enables recompression, which prior approaches for multi-granularity memory tagging were unable to transparently perform [131, 161, 127]. When cache lines are evicted from on-chip caches, the bit can be updated to "recompress" the corresponding memory chunk. We will refer to this bit as the non-default metadata (NDM) bit, which will be used to determine whether a memory chunk contains only default metadata values (0) or at least one non-default value (1). We extended the baseline metadata management algorithm [100] to allocate NDM bits in shared memory adjacent to the corresponding metadata. In our evaluation, a page of allocated metadata requires 64 NDM bits for the chunks contained within the page, and a page of NDM bits can hold mappings

for 512 4KB pages of metadata.

Metadata caches can use NDM bits to filter out metadata accesses to default metadata values in one of two possible ways. To take advantage of NDM bits, we modified the cache controllers in the metadata cache hierarchy using one of two possible approaches. These approaches can be distinguished by the level of the cache hierarchy at which they operate at and how transparent they are to the cache.

The first approach, which is shown in Figure 5.6, is named NDM with Eager Read (NDM-E) and will ”eagerly” read the NDM bit of a metadata memory chunk before the L1 cache is read. The NDM bits are stored in a cache structure that is organized almost identically to the tag TLB. On each read, the NDM-E cache returns a bit–the NDM bit–corresponding to the looked up address. If the NDM bit of the accessed line is 0, indicating that the cache block contains only default metadata values, then the cache read can be avoided. For cache writes in NDM-E, the cache controller must first check the NDM bit of the written line. If the bit is not set, then the first-level cache must first allocate a line for the address. The allocation performed will also write the default metadata value for each value contained within the cache line. After the write completes, the NDM bit must be set so as to ensure that future reads from the cache line are appropriately directed to the metadata cache hierarchy. For cache write-backs in NDM-E, the written-back line will be checked for recompression. The second-level cache will not be written if the line can be recompressed.

The second approach, which is shown in Figure 5.7, is named NDM and is more transparent to the first-level caches than NDM-E. The NDM cache will only be read if a metadata access misses in the last-level cache. If the NDM bit

for the missed line is 0, then the cache controller will avoid the memory read and allocate a cache line in the last level with all zeros. However, if the NDM bit for the missed line is 1, then the access must be fulfilled from memory. Cache writes in NDM are performed the same way as conventional caches because the existence of NDM bits only affect off-chip memory accesses. Recompression is performed on write-backs to memory.

NDM-E offers the advantage of more efficient usage of on-chip caches by filtering out accesses that bring cache lines with default metadata values into the on-chip caches. However, because NDM-E must access the NDM bits on each L1 metadata cache access, it suffers from potentially higher energy consumption. If the NDM bit is checked before the L1 cache is read, NDM-E can also increase the delay of the first-level cache. In the evaluation, we conservatively model this as an additional cycle of delay.

**Compressed Metadata**

To take advantage of the frequent value locality observed in Section 5.1, we also evaluated an approach that leverages value-based compression named dynamic metadata compression (DMC). DMC attempts to compress double-word values in a metadata cache line using a small dictionary and caching dictionary indices when a cache line is found to be compressible. The example in Figure 5.10 illustrates when a cache line is compressible in this scheme. This optimization can utilize on-chip cache space more efficiently given the fulfillment of two main criteria. The first is that the dictionary is large enough to hold the most frequently occurring values observed in the metadata working set. The second is that an algorithm is employed that can effectively populate the dictionary with

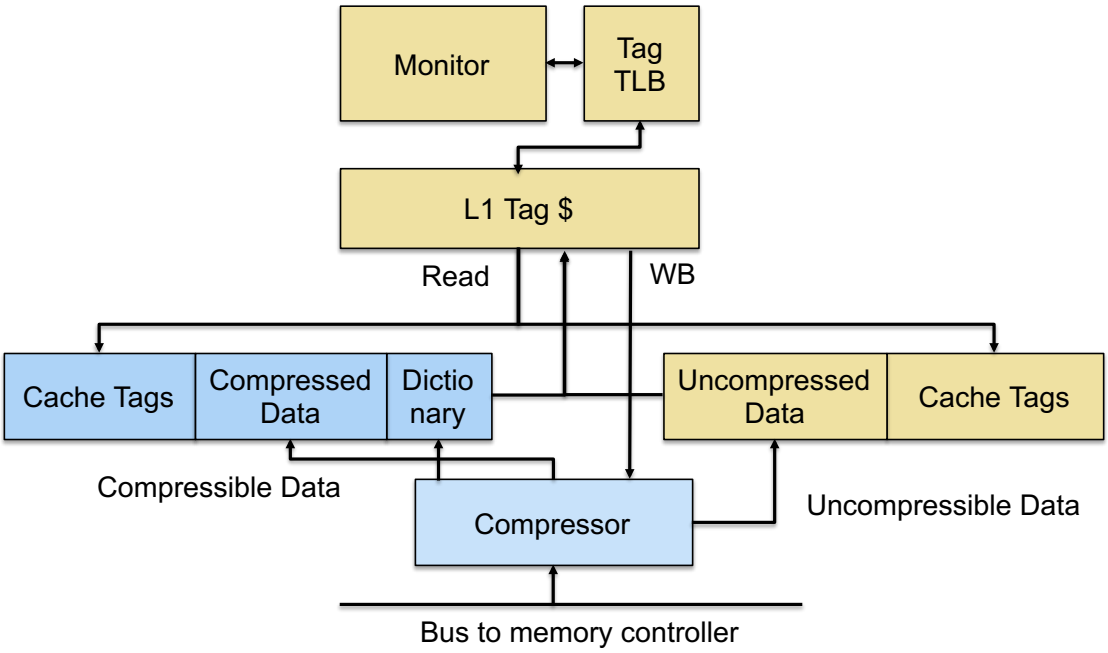



Figure 5.8: Block diagram of the DMC cache compression scheme for metadata in the on-chip last-level cache. The last-level cache in this scheme is split into two halves that are equal in area in order to store compressed and uncompressed data.

useful values. From the locality study of the benchmarks used in our evaluation, we determined that a dictionary of 32 entries was sufficient, which also allows metadata values to be compressed by a factor of 12.8X.

Previously proposed software-based approaches for frequent-value compression on the main processing core are able to populate dictionary values through values observed during short profiling phases [163]. This approach is not applicable when the accesses are performed transparently by accelerator hardware. DMC includes a simple hardware approach that caches values found in cache traffic in an attempt to dynamically find values to use for compression. This approach uses a candidate table, which tracks the frequency of occurrence for each metadata value observed at runtime using a count that is read and in-

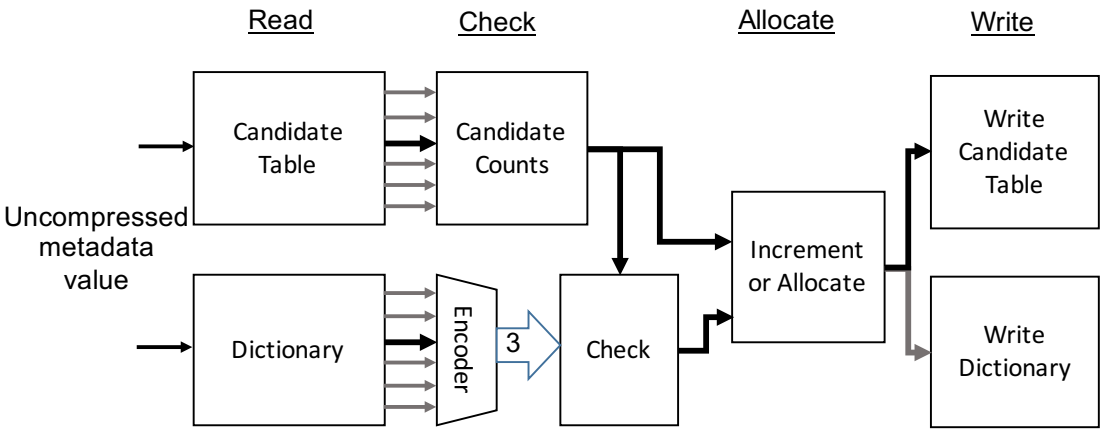



Figure 5.9: Block diagram of the design for the DMC compression logic. The compression data path is used during cache refills and writebacks to compress values in the cache line. Even if the line is not compressible, the candidate table is updated to track the frequency of occurrence of metadata values. When a value has been observed more than a certain number of times, it is written to the dictionary and used for compression.

Dictionary:

| Index | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| Value | 0 | 9C6B86009C6C848 | FFE74170FFE742FF | 9C83EA809C84E90 |

Uncompressed Cache Line:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 9C6B86009 C6C848 | 0 | 0 | 9C83EA809 C84E90 | 0 | FFE74170F FE742FF | 0 |

Compressed Cache Line:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00 | 01 | 00 | 00 | 11 | 00 | 10 | 00 |

Figure 5.10: Example of compression for an uncompressed cache line of eight values and a dictionary with four values.

cremented each time the value is observed. When the count reaches a threshold, then the value will be copied to the dictionary for use in compression. We empirically determined 64 as the threshold for our evaluation by testing a range of threshold values. We found that smaller thresholds can result in the dictionary becoming populated with values that were not the most frequently occurring values for certain benchmarks; on the other hand, larger thresholds can result in too few values being written into the dictionary. For many benchmarks, small thresholds were sufficient as they resulted in the most cache hits for compressed data. The effectiveness of small thresholds also reflects the frequent value locality observed in Section 5.1.

On each write-back from the first-level cache and on a refill from memory, each value in the cache line is evaluated in a hardware pipeline that compares the value against the dictionary and candidate table entries. The pipeline can be broken down into four high-level stages: read, check, allocate, and write. In the read step, the uncompressed value in the cache line is used to index the candidate table and the dictionary table, which can be built using a CAM array [107] that asserts the index of the matching entry. In the check step, the dictionary match determines whether the value is compressible; if the value is not compressible, then the cache line is not compressible. The index of the matching candidate table entry is used to access the candidate count array for the candidate count. In the allocate step, the dictionary will be written with the uncompressed value if there was a candidate table hit, the corresponding candidate count reached a saturated value, and the dictionary missed. If there was a miss in the candidate table, then the candidate table replacement algorithm will be used to find the candidate table entry in which the new value will be written. In the write step, the candidate table is written, the candidate count

array is written with the incremented count, and the dictionary write (if any) is performed.

For replacement, the candidate table uses the pseudo-LRU policy, and the dictionary uses FIFO. We also evaluated Full-LRU and FIFO replacement policies for the candidate table and found that each policy performed similarly in terms of the values that were written into the dictionary. For a subset of the benchmarks, pseudo-LRU performed slightly better than FIFO in terms of the number of cache lines that were compressed. The dictionary is populated using FIFO replacement until it becomes full. To obviate the need to replace all corresponding compressed values when a dictionary entry is replaced, we block all writes to the dictionary (and clock gate the compression hardware) after the dictionary becomes full. However, between long-running phases of a program, the set of metadata values that are useful for compression may be different. To adapt to the difference in behavior between phases, a mechanism is needed to age dictionary entries; this can be performed by periodically clearing the dictionary and cache and re-training both arrays [163, 124]. In our evaluation, we modeled the periodic clearing of the arrays by re-training each at the beginning of a benchmark.

The remainder of this section will discuss how DMC caches compressed and uncompressed data. For all cache lines that enter the last-level cache, the cache controller will read the contents of the dictionary and attempt to match each value to an existing entry using the compression pipeline. When a value matches a dictionary entry, it is possible to compress the value, and the matching dictionary index is buffered in a compressed line. If all metadata values in an incoming cache line are compressible, then the line can be cached in com-

pressed form. Otherwise, the original uncompressed cache line must be cached.

In DMC, both forms of data can be stored by separating the compressed and uncompressed caches into equally-sized halves in terms of physical size. By compressing incoming values into smaller dictionary indexes, the compressed cache requires fewer data bits and can use the area saved by compression to have a larger number cache lines. [1] When writing to the second level cache, after the incoming line has been checked for compressibility, the compressed line is steered towards the compressed cache, and an invalidation signal is sent to the uncompressed cache for the same address. Otherwise, the cache line is steered to the uncompressed cache, and an invalidation is sent to the compressed cache. The invalidations are necessary to maintain mutual exclusion.

In operation, the compressed and uncompressed portions of the cache are accessed in parallel for reads. Reads that hit in the uncompressed cache return the data directly. Reads that hit in the compressed cache will decompress each value that is stored in the compressed line to reconstruct the uncompressed cache line; this incurs a hit-delay penalty proportional to the number of values in the cache line. To perform decompression, the dictionary is replicated and read for decompression. Although the dictionary for compression must be organized as a fully-associative array for matching purposes, the replica can be a simpler SRAM array that is addressed linearly. Compressed values in the line are used in sequence to index the dictionary replica to find the uncompressed value. Once the original uncompressed cache line is reconstructed, the line is sent to the upper-level cache to complete the read. We modeled the delay for performing decompression based on the assumption that a new value can be read from the dictionary in each cycle.

[1] Using estimates from Cacti, a compressed cache can hold 4X more lines.

## 5.3 Evaluation

### 5.3.1 Methodology

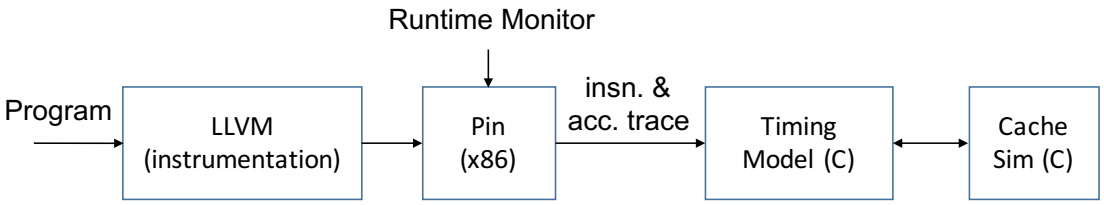



Figure 5.11: The evaluation framework used for the cache designs.

| | Main Core | Conventional Base | NDM-L | NDM-E | Compressed |
|---|---|---|---|---|---|
| L1 Size | 6-Way, 24KB | 2-Way, 4KB | | | |
| L2 Size | 8-Way, 512KB | 8-Way, 128KB | | | |
| L1 Delay (cycles) | 3 | 1 | 1 | 1 (NDM hit) or 2 (NDM miss) | |
| L2 Delay (cycles) | 15 | 8 | 8 | 8 | 8 |
| Compression Delay | - | | | | 8 cycles |
| Decompression Delay | - | | | | 8 cycles |
| TLB Size (entries) | 16 L1, 64 L2 | | | | |
| Block Size (bytes) | 64 | | | | |
| Memory Delay (cycles) | 200 | | | | |
| Benchmarks | SPEC2K: art bzip2 crafty equake gcc gzip mcf mesa parser perlbmk twolf vpr<br>SPEC2K6: gcc gobmk h264ref hmmer libquantum mcf milc perlbench | | | | |

Table 5.2: Simulation parameters and benchmarks.

To evaluate the performance of metadata cache organizations, we used trace-driven simulation and estimated the performance of monitored applications with a simple (and conservative) timing model. The high-level flow of the simulation framework is shown in Figure 5.11. The runtime monitors are implemented as analysis tools for the Pin [70] binary instrumentation framework; and monitored applications are run with Pin to capture their behavior. During the course of execution of each monitored application, instruction counts, memory addresses, and accessed metadata values are captured by the analysis tools and stored to trace files. To cope with finite amounts of storage, the

traces contain 1.5 billion metadata accesses for each application and monitoring approach. The traces are then evaluated with a timing model that models the cache and memory delays in more detail.

In the timing model, absent of stalls, the main processing core retires one (non-memory) instruction in each cycle and forwards the retired instruction to the runtime monitor using the Core-Fabric FIFO queue. Each runtime monitor reads retired instructions from the queue and will access its own cache hierarchy for the metadata of memory operands. When an access misses the first-level cache, the core or monitor will stall and wait for the request to be fulfilled from the lower levels. Table 5.2 summarizes the other parameters in the evaluation. Latencies for the core caches were gathered from published numbers for the Atom processor [35] and metadata cache latencies were collected from running Cacti 6.5 [93].

To amortize the effects of initial transient behavior, we warmed up for one billion accesses and collected statistics for the remainder of the traces using the performance model. In additional, to get a better sense of the effectiveness of DMC, we also compared its performance to an oracle approach that can populate the dictionary using values that are profiled from each trace. In this oracle approach, the dictionary for each benchmark is statically derived from profiling the entirety of each trace and the same dictionary is used for compression for the duration of the simulation.

### 5.3.2 Performance

Figures 5.12(a), 5.12(b), and 5.12(c) show the CPI of monitored applications normalized to a baseline where monitoring is off. From left to right, the graphs show the effect on CPI of metadata cache organizations that use conventionally organized metadata L2, NDM with a 128KB metadata L2, NDM-E with a 128KB metadata L2, NDM-E with DMC that can cache 1K compressed lines and 1K uncompressed lines, and NDM-E with cache compression using oracle knowledge. To better differentiate the sources of performance overheads and how the optimization affects each source, the CPI is broken down by the baseline application performance, stalls from memory contention, and stalls from queue wait. Memory contention reflects increases in delay for memory accesses from the main processing core due to bandwidth contention with accesses from the runtime monitor. Queue wait reflects cycles in which the main processing core must stall because the queue between the main processing core and the runtime monitor is full. The results show that queue wait accounts for the majority of the performance overheads.

When comparing the performance across the evaluated monitoring techniques that use a conventionally organized 128KB L2 metadata cache, IFC has the highest slowdown at 3.6x. Unlike FBC and FST, which can filter more instructions that are not relevant to the monitoring performed, IFC has higher slowdowns because it must monitor all memory accesses. By filtering memory accesses that are not performed at a word granularity, FBC has lower overheads than IFC at 2.2x. Lastly, because FST was customized for a small subset of the x86 instructions that are relevant for security monitoring, it can filter out a larger percentage of instructions and accesses and has the lowest performance

overheads of the three at 1.9x.

For a cache hierarchy using a 128KB L2, adding a small cache of NDM bits dramatically reduces the performance overheads of FST, while the performance overheads of FBC and IFC are also improved. Compared to a 128KB L2, NDM reduces the performance overheads of FST by 39.3%, FBC by 15.5%, and IFC by 21.5%. For all three monitoring approaches, the small 128KB metadata L2 cache with NDM is able to attain performance that is comparable to using a much larger (1MB) conventionally organized L2 metadata cache. Compared to NDM, NDM-E is able to provide an additional 3% reduction in overheads for FBC, 13% for IFC, and 8.9% for FST. When NDM-E is enabled, it can take full advantage of the sparsity of FST metadata to reduce the performances to less than 3% on average compared to when no monitoring is performed.

Lastly, the combination of NDM-E and DMC can further reduce the average performance overheads by all three monitoring approaches and outperform a much larger 1MB L2 conventional cache. Compared to NDM-E alone, the combination of optimizations are able to reduce the performance overheads of FBC by 24.8% and IFC by 13.3%. FST, whose performance overheads were already very low with NDM-E, improves by another 1.0% when NDM-E and DMC are both used. Compared to using a much larger 1MB conventional L2, NDM-E and DMC improve the performance of FBC by 27%, FST by 35.2%, and IFC by 22.2%. Compared to when non monitoring is performed, the performance overheads of NDM-E and DMC are reduced from 117% to 33% for FBC, from 267% to 117% for IFC, and from 86% to less than 3% for FST. We will also note that for all three monitoring approaches, while the oracle approach for populating the dictionary for compression outperformed DMC on average, the difference

in performance was very small (less than 1%), indicating that it is almost as effective as profiling for finding the most frequently occurring metadata values for these scenarios.

### 5.3.3 Metadata Access Delays

To better understand the effects of each of the optimizations, we also evaluated the average latencies for metadata accesses made by runtime monitoring approaches. Figures 5.13(a), 5.13(b), and 5.13(c) show the average metadata access latencies for FBC, FST, and IFC with increasing conventional L2 metadata caches sizes and with the proposed optimizations. For each of the evaluated monitoring approaches that uses conventionally organized L2 metadata caches, the delays were dominated by the contribution of metadata accesses that missed to memory.

When NDM is enabled for FBC and IFC with a 128KB L2 metadata cache and is used to filter memory traffic, the number of metadata accesses that miss to memory is reduced to levels that are comparable to a much larger 1MB L2 metadata cache. For FST, NDM filters a much larger percentage of memory traffic, which helps to reduce its performance overheads. When NDM-E is enabled for FST and IFC, a large percentage of metadata accesses are diverted from both the L1 and L2 metadata caches to the NDM cache. This helps to reduce the pressure on the small metadata caches and further reduces the average delay and the performance overheads of these monitoring approaches.

Lastly, NDM-E with DMC is able to further improve over NDM-E by virtually increasing the capacity of the L1 caches for FBC and IFC and reduce accesses

to L2. Compared to NDM-E, DMC can convert more of the metadata accesses that miss to memory to L2 metadata cache hits for FBC and IFC.

### 5.3.4 Area and Power Overheads

| Component | Capacity | Area ($um^2$) | Energy/Access (pJ) | Static Energy (mW) |
|---|---|---|---|---|
| L1 Cache | 4KB | 348,553.2 | 25.6 | 78.2 |
| L2 Cache | 64KB | 296,855 | 28.4 | 7.98 |
| | 128KB | 417,439.4 | 43.2 | 15.4 |
| | 256KB | 666,477.5 | 72.9 | 30.4 |
| | 512KB | 1,227,341.7 | 105.1 | 58.3 |
| | 1MB L2 | 2,379,850 | 175.7 | 115.3 |
| Compressed L2 | 20KB | 100,873.5 | 23.6 | 3.8 |
| DMC | 64KB+64KB | 39,7728.5 | 52.0 | 11.78 |

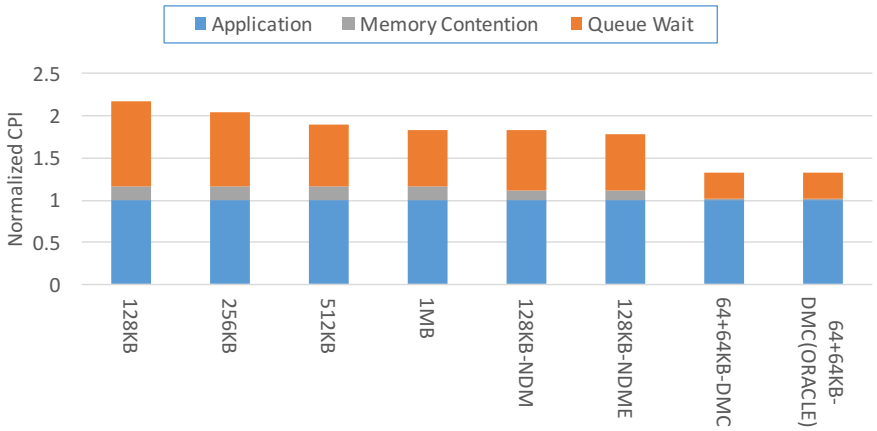
Table 5.3: Area and power of evaluated cache sizes for the 65nm technology generation

In this section, we will discuss the area and power overheads of the NDM and DMC; these overheads are low by design. In addition, we discuss these overheads within the context of the area of the metadata caches, which have area and power estimates shown in Table 5.3. The overheads were estimated using Cacti [93].

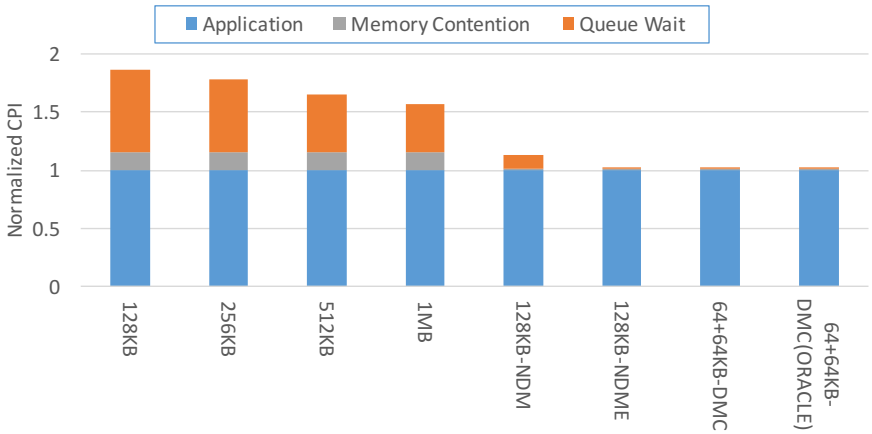
For NDM and NDM-E, the primary area and power overheads stem from the additional on-chip arrays for storing 64 NDM bits per memory page. Because NDM-E is designed to be accessed alongside the metadata TLBs, the area can be kept small by re-using the TLB's tag matching and decode logic to select the corresponding NDM word. For a fully associative TLB with a CAM array for tag matching and a SRAM array for holding the cached data, the addition of the SRAM array for NDM-E will add minor overheads. Prior research has

shown that the area of such caches is dominated by the CAM arrays, which take up almost 5x as much area as an equivalent SRAM array [50]. Although NDM-E introduces static power overheads, the reduction in metadata cache accesses for NDM-E can offset the dynamic power overheads. For NDM, the area overheads can be greater, since NDM would require a duplication of the structure resembling the metadata TLB. However, NDM can still provide significant energy savings by avoiding metadata accesses that miss to memory.

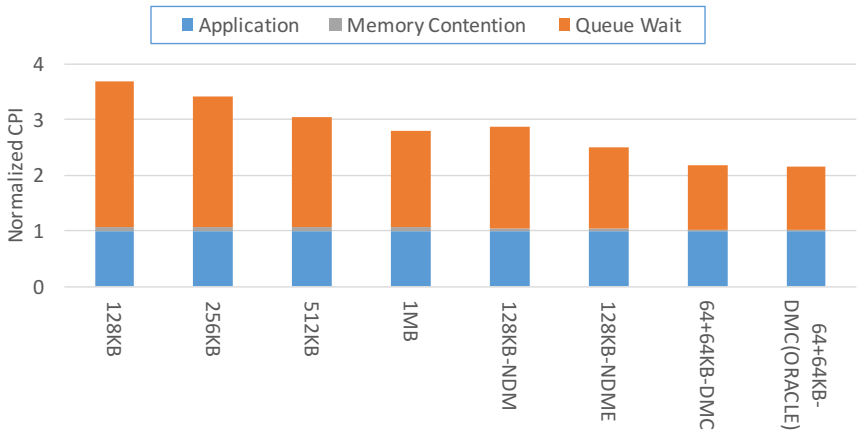
For DMC, the area overheads can be primarily attributed to the candidate and dictionary arrays used for the compressor. A straightforward implementation of the compression logic using 32x32 CAMs [66]–two for the dictionary, and eight for the candidate array–would result in an area overhead of roughly 230,000$\mu$m$^2$. Table 5.3 also shows that the additional cache used in DMC for storing compressed data would consume another 100,000$\mu$m$^2$. However, when compared to a much larger 1MB conventionally-organized L2 cache, which DMC would out-perform, the compression logic and DMC caches are nearly 4x smaller. In terms of energy and power, although conventional CAM designs implemented using NOR matching cells would consume significant power, these overheads can be reduced by either 1) implementing the CAM array using a NAND topology [107] (at the cost of longer compression latencies); or 2) by simplifying the design to use less associative arrays (at the cost of possibly using sub-optimal values for compression).




(a) FBC

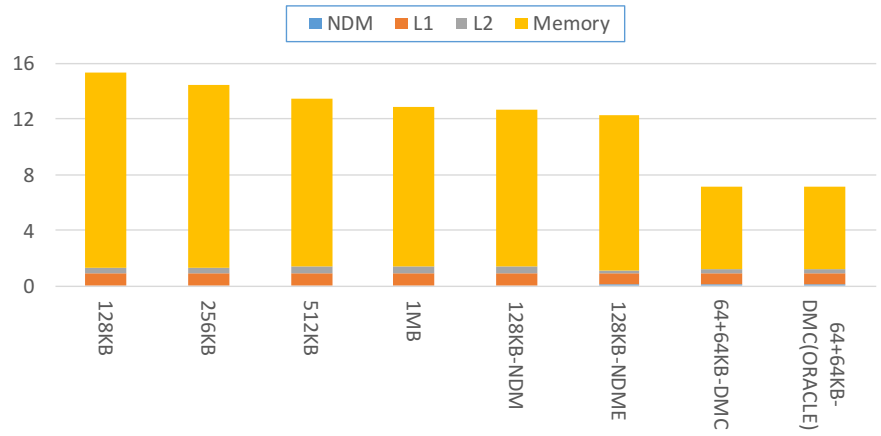



(b) FST

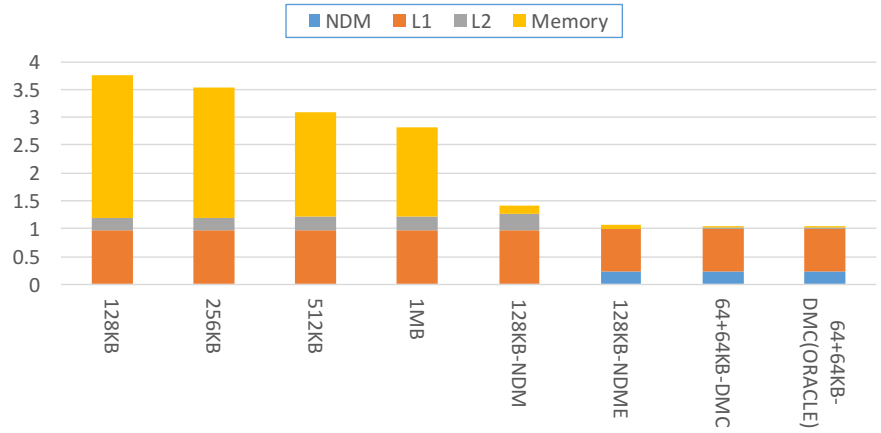



(c) IFC

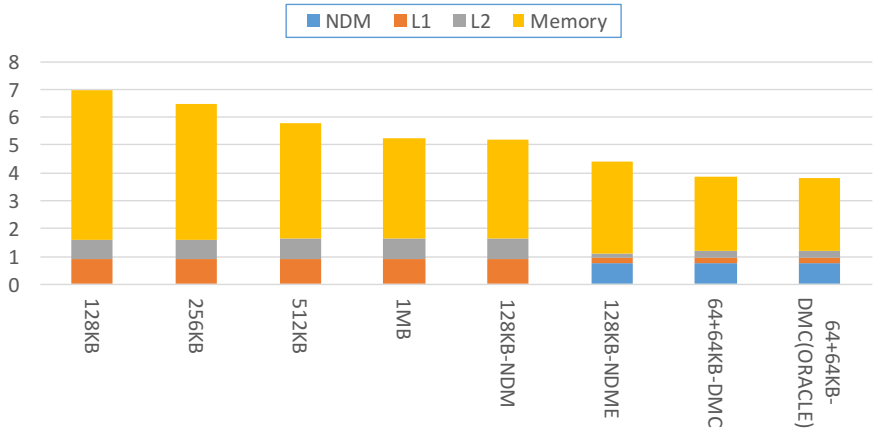
Figure 5.12: Normalized performance of applications with runtime monitoring enabled across different metadata cache sizes, NDM, NDM-E, DMC, and DMC with oracle dictionary. The same 4KB L1 metadata cache is used for each configuration.




(a) FBC




(b) FST




(c) IFC

Figure 5.13: Breakdown of metadata access latencies and contributions of hits at different levels of the metadata memory hierarchy with runtime monitoring enabled across different metadata cache sizes, NDM, NDM-E, DMC, and DMC with oracle dictionary.

## 5.4 Summary

In this chapter, we evaluated optimizations for the metadata memory hierarchy of a Harmoni-based system for runtime monitoring that can transparently perform monitoring using large medata. To motivate the optimizations, we analyzed the characteristics of metadata for several runtime monitoring approaches that use large tags and found two distinct characteristics of the metadata used for bookkeeping and checking. In particular, we found that metadata tends to be sparse and take on a small handful of unique values over the course of typical applications. To take advantage of these characteristics, we proposed two lightweight hardware optimizations on the metadata cache hierarchy: NDM cache to take advantage of the sparsity, and DMC to take advantage of the value locality. The evaluation of the proposed approaches showed that NDM allows a metadata cache hierarchy using a 128KB L2 to perform similarly to a system that uses a conventional L2 cache that is 8x larger and the combination of NDME and DMC can outperform a system that uses a conventional 1MB L2 cache by 22.2%-35.2%. Overall, the NDME and DMC can reduce the performance overheads of runtime monitoring with large tags from 86-267% to 3-117% of a baseline design that uses conventional caches.

# CHAPTER 6

# SUMMARY

Runtime monitoring is a promising approach for security and reliability. However, whereas both flexibility and efficiency are important for runtime monitoring implementations, there is currently a tradeoff between these characteristics. In this thesis, we built a framework for runtime monitoring that can bridge this gap between flexibility and efficiency. The FlexCore and Harmoni architectures demonstrated how to leverage reconfigurable hardware for mapping hardware implementations runtime monitoring functions with low performance overheads on monitored applications. In this chapter, we will summarize the key contributions of this thesis.

In Chapter 3, we introduced an initial implementation of the runtime monitoring framework named FlexCore. The primary contribution of the work in this chapter was showing how an island-style, bit-level reconfigurable fabric can be employed to achieve the desired characteristics of both flexibility and efficiency for runtime monitoring. Using a reconfigurable fabric that had 50% of the area of the baseline processor, FlexCore had the flexibility to implement a broad spectrum of monitoring functions ranging from simple memory checks to more in-depth checking of computation in the monitored application. By implementing the runtime monitoring features to bit-level reconfigurable hardware, the approach improves in efficiency in comparison with software-only and multiprocessor implementations of runtime monitoring approaches. While a custom-hardware based implementation is still better in terms of performance, the FlexCore architecture manages to close the gap for embedded applications by filtering instructions that are not relevant to the monitoring being performed

and selectively incorporating dedicated hardware units in the reconfigurable fabric. Overall, we were able to limit the performance overheads of runtime monitoring to less than 20% for a majority of evaluated approaches.

However, for runtime monitoring approaches that are not able to keep pace with the frequency of the main processing core when mapped to the reconfigurable fabric, the performance overheads of runtime monitoring on FlexCore became more significant. To work through these limitations, we explored two methods to reduce the performance overheads of the runtime monitoring framework.

Harmoni was proposed in Chapter 4 that replaces the island-style bit-level reconfigurable fabric of FlexCore with a more customized accelerator that is more purpose-built for tagging. The primary contribution of the work in this chapter was showing how many runtime monitoring approaches are built on the notion of tagging, where software correctness problems are tracked and verified using metadata that shadows data in the application. By taking advantage of this observation, we are able to restrict the programmability of the co-processor without constraining the important classes of monitoring approaches that it is able to implement. Harmoni is able to achieve much higher frequencies by using a more custom hardware to manipulate the metadata, while retaining flexibility by still being able to implement most of the example monitoring approaches evaluated on FlexCore. The higher throughput of Harmoni reduces its performance overheads to less than 10% for processing core that can run at a few GHz. Further, the increased frequency and throughput of Harmoni allows it to keep pace with larger, higher performance, and more power hungry processing cores.

Monitoring approaches built on the notion of tagging must also access and manage metadata in shared main memory. When the metadata is more than a few bits in size, accessing the metadata can become a more significant source of performance overheads and hardware cost. In Chapter 5, we studied the characteristics of metadata maintained by several example monitoring approaches that can transparently manage and use larger metadata values. Through the study, we found that the metadata maintained by these approaches exhibited the characteristics of sparsity and frequent value locality. The primary contribution of the work in this chapter was showing how these characteristics can be leveraged to more efficiently cache metadata in small on-chip caches and to reduce the memory bandwidth used to move metadata between on-chip caches and shared main memory. To take advantage of sparsity, we proposed an NDM cache, which can filter and compress metadata accesses for default values. For frequent value locality, we proposed DMC, an approach that uses dynamic dictionary-based compression to reduce the in-cache representation of values stored in the last-level metadata cache. The combination of approaches are more lightweight than increasing the size of the last-level cache in the metadata cache hierarchy. Our evaluation of the approaches, which are orthogonal and can easily be combined, showed that they can dramatically reduce the number of metadata cache accesses that miss to memory and reduce the performance overheads of runtime monitoring with large metadata values from 86-267% to 3-117%.

Going forward, as hardware designs and the correctness issues they address increase in complexity, greater automation and faster implementation can be vital for the effectiveness of the designs shown in this work. Because many of the examples approaches shown in this work were interpreted from prior art

and manually implemented and optimized for the monitoring frameworks, avenues of future work are for compilation tools to automatically map software implementations of runtime monitoring approaches (implemented on debugging frameworks such as Valgrind [122] or Pin [70]) onto the co-processing fabrics proposed in this work.

## BIBLIOGRAPHY


[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, 2005.

[2] Lucas Ackerman. Field extensible microarchitecture: AMBA memory interface and software- controlled tags. *Master of Engineering Design Project Report*, 2012.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[5] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 212–, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Alaa R. Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, Wisconsin, 2004.

[7] Altera. Robust SEU mitigation with Stratix III FPGAs. WP-01012-1.0.

[8] Brad Arkin. Important customer security announcement. `http://blogs.adobe.com/conversations/2013/10/important-customer-security-announcement.html`, October 2013.

[9] ARM. ARM926EJ-S technical reference manual revision: r0p5, 2008.

[10] Todd Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32$^{th}$ International Symposium on Microarchitecture*, November 1999.

[11] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, 1994.

[12] Algirdas Avizienis. The N-Version approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, December 1985.

[13] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, Monica S. Lam, V. Benjamin, Livshits Monica, and S. Lam. Improving software security with a c pointer analysis. In *In ICSE 05: Proceedings of the 27th International Conference on Software Engineering*, 2005.

[14] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, 2000.

[15] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, 2003.

[16] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, 2006.

[17] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, 2003.

[18] K. J. Biba. Integrity considerations for secure computer systems. MITRE Technical Report TR-3153, April 1977.

[19] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, 2000.

[20] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. AAI0807735.

[21] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack, 10 (56)*, May 2000.

[22] Michael Burrows, StephenN. Freund, and JanetL. Wiener. Run-time type checking for binary programs. In *CC 2003*, pages 90–105, April 2003.

[23] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[24] David Chen, Enoch Peserico, Larry Rudolph, and Sma Fellow. A dynamically partitionable compressed cache. In *In Proceedings of the Singapore-MIT Alliance Symposium*, 2003.

[25] Liming Chen and Algirdas Avizienis. N-Version Programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing FTCS-25*, Jun 1995.

[26] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R.K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, June 2005.

[27] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 63–65, New York, NY, USA, 2006. ACM.

[28] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society.

[29] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, 2005.

[30] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Trans. Very Large Scale Integr. Syst.*, 18(8):1196–1208, August 2010.

[31] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[32] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-Hardware Architecture for Self-Protecting Data. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[33] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective Memory Protection Using Dynamic Tainting. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, 2007.

[34] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[35] Intel Coporation. Intel Atom Processor Z510, 2008.

[36] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. DISE: A programmable macro engine for customizing applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, 2003.

[37] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM.

[38] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. *Pointguard™*: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, 2003.

[39] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of

buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[40] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *In Proceedings of the 15th USENIX Security Symposium*, 2006.

[41] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37$^{th}$ International Conference on Microarchitecture*, December 2004.

[42] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 235–248, New York, NY, USA, 2005. ACM.

[43] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[44] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *ASPLOS XIII: Proceedings of the 13$^{th}$ international conference on Architectural support for programming languages and operating systems*, pages 103–114, 2008.

[45] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and Andre DeHon. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[46] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, 2015.

[47] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[48] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, 2003.

[49] Julien Dusser and André Seznec. Decoupled zero-compressed memory. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 77–86, New York, NY, USA, 2011. ACM.

[50] Burak Erbagci, Fangfei Liu, Cagla Cakir, Nail Etkin Can Akkaya, Ruby Lee, and Ken Mai. A 32kb secure cache memory with dynamic replacement mapping in 65nm bulk CMOS. In *IEEE Asian Solid-States Circuit Conference (A-SSCC)*, 2015.

[51] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[52] Justin H. Ferguson. Understanding the heap by breaking it. `https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf`, 2007.

[53] Edward A. Feustel. On the advantages of tagged architecture. *Computers, IEEE Transactions on*, C-22(7):644 –656, july 1973.

[54] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[55] Sotiria Fytraki, , Evangelos Vlachos, Onur Kocberber, Babak Falsafi, and Boris Grot. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *HPCA 2014*, pages 108–119, Feb 2014.

[56] Jiri Gaisler, Edvin Catovic, Marko Isomaki, Kristoffer Glembo, and Sandi Habinc. GRLIB IP Core User's Manual, 2008.

[57] Tal Garfinkel. Traps and Pitfalls: Practical problems in system call interposition based security tools. In *In Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[58] Edward F. Gehringer and J. Leslie Keedy. Tagged architecture: how compelling are its advantages? In *Proceedings of the 12th annual international symposium on Computer architecture*, ISCA '85, pages 162–170, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[59] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, 1996.

[60] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *Annual IEEE International Workshop on Workload Characterization*, 2001.

[61] Erik G. Hallnor and Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, WMPI '04, pages 9–15, New York, NY, USA, 2004. ACM.

[62] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, 2002.

[63] Samuel P Harbison and Guy L. Steel Jr. *C: A Reference Manual (3rd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[64] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in c and c++ programs. In *Proceedings of the Winter USENIX Conference*, pages 125–138, 1992.

[65] J.R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12 –21, April 1997.

[66] Po-Tsang Huang, Wei-Keng Chang, and Wei Hwang. Low power content addressable memory with pre-comparison sceheme and dual-vdd tech-

mique. *Department of Electronics Engineering & Institute of Electronics, and Microelectronics and Information Systems Research Centre, May2010*, 2006.

[67] Ruirui Huang, Daniel Y. Deng, and G. Edward Suh. Orthrus: Efficient software integrity protection on multi-cores. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[68] IBM. ProPolice detector. `www.trl.ibm.com/projects/security/ssp/`.

[69] Synopsys Inc. Designware digital ip quick reference guide, December 2011.

[70] Intel. Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[71] https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[72] Todd Jackson, Babak Salamat, Gregor Wagner, Christian Wimmer, and Michael Franz. On the effectiveness of multi-variant program execution for vulnerability detection and prevention. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, 2010.

[73] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *In Proceedings of the Network and Distributed Systems Security Symposium*, 1999.

[74] Peter Andrew Jamieson and Jonathan Rose. Enhancing the area efficiency of fpgas with hard circuits using shadow clusters. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 1–8, Dec 2006.

[75] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, 2002.

[76] Jos Joao, Onur Mutlu, and Yale Patt. Flexible reference-counting-based

hardware acceleration for garbage collection. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[77] Richard W M Jones and Paul H J Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *in Distributed Enterprise Applications. HP Labs Tech Report*, pages 255–283, 1997.

[78] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 105 –114, 29 2009-july 2 2009.

[79] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, 2003.

[80] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference*, pages 423–430, Sep 1996.

[81] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, 2006.

[82] Ian Kuon and Jonathan Rose. Area and Delay Trade-offs in the Circuit and Architecture Design of FPGAs. In *Proceedings of the 2008 ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays*, 2008.

[83] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[84] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 184–191, 1999.

[85] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *J. Syst. Archit.*, 46:1365–1382, December 2000.

[86] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, April 2003.

[87] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. *SIGSOFT Softw. Eng. Notes*, 2001.

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[89] Bruce J. MacLennan. *Principles of Programming Languages (3rd Edition): Design, Evaluation, and Implementation*. Oxford University Press, Inc., 1999.

[90] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the* $40^{th}$ *Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[91] Albert Meixner and Daniel J. Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, 2007.

[92] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, 1985.

[93] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman Jouppi. CACTI 6.0: A Tool to Model Large Caches. HP Labs. In *Tech Report HPL-2009-85*, 2009.

[94] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 211–221, New York, NY, USA, 2008. ACM.

[95] Santosh Nagarakatte, Milo Martin, and Steve Zdancewic. Hardware-enforced comprehensive memory safety. *IEEE Micro*, May 2013.

[96] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve

Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[97] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[98] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, 2002.

[99] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-Checking Entire Programs without Recompiling. In *SPACE 2004*, 2003.

[100] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, 2007.

[101] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[102] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the* 2005 *Network and Distributed Systems Symposium*, February 2005.

[103] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, and David Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[104] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[105] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, Mar 2002.

[106] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[107] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 2006.

[108] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw. Pract. Exper.*, 27(1), January 1997.

[109] The PaX Team. The PaX project. `http://pax.grsecurity.net/`.

[110] perlsec. Perl documentation. `http://perldoc.perl.org/perlsec.html`.

[111] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, 2003.

[112] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39(5):235–248, October 2005.

[113] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[114] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, 2004.

[115] Jon A. Rochlis and Mark W. Eichin. With Microscope and Tweezers: The Worm from MIT's Perspective. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, May 1989.

[116] Benjamin Rodes. Stack Layout Transformation: Towards diversity for

securing binary programs. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, 2012.

[117] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the* $11^{th}$ *Annual Network and Distributed System Security Symposium*, pages 159–169, February 2004.

[118] Babak Salamat, Andreas Gal, and Michael Franz. Reverse stack execution in a multi-variant execution environment. In *In Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[119] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, 2009.

[120] Somayeh Sardashti and David A. Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 62–73, New York, NY, USA, 2013. ACM.

[121] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[122] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[123] A. Seznec. Decoupled sectored caches: Conciliating low tag implementation cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 384–393, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[124] Andr Seznec and Pierre Michaud. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism*, 2006.

[125] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50(2/3):261–275, March 2006.

[126] Weidong Shi, J.B. Fryman, Guofei Gu, H.-H.S. Lee, Youtao Zhang, and Jun Yang. InfoShield: a security architecture for protecting information usage in memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, Feb 2006.

[127] R. Shioya, Daewung Kim, K. Horio, M. Goshima, and S. Sakai. Low-overhead architecture for security tag. In *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, pages 135 – 142, nov. 2009.

[128] INC SPARC International. The SPARC Architecture Manual Version 8, 1992.

[129] Peter Steenkiste and John Hennessy. Tags and type checking in lisp: hardware and software approaches. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, ASPLOS-II, pages 50–59, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[130] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, 2003.

[131] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 85–96, New York, NY, USA, 2004. ACM.

[132] Moran Surf and Amichai Shulman. How safe is it out there? *Imperva Inc.*, 2004.

[133] Symantec. Symantec internet security threat report trends for january 06june 06. `http://www.symantec.com/specprog/threatreport/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf`, 2006.

[134] Synopsys. Synopsys(r) FPGA synthesis products D-2010.03, 2010.

[135] Synopsys. Design compiler version F-2011.09-SP3, 2011.

[136] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR Lisp Architecture. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, 1986.

[137] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 94–105, Washington, DC, USA, 2008. IEEE Computer Society.

[138] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Bita Mazloom, Shashidar Mysore, Frederic T. Chong, and Timothy Sherwood. Gate-Level Information-Flow Tracking for Secure Architectures. *IEEE Micro*, January 2010.

[139] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT). *IBM J. of Research and Development*, 2001.

[140] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 115–128, New York, NY, USA, 2007. ACM.

[141] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Workshop on Binary Instrumentation and Application*, San Jose, CA, October 2007.

[142] US-CERT. Sb13-140. `http://www.us-cert.gov/ncas/bulletins/SB13-140`, May 2013.

[143] US-CERT/NIST. Cve-2013-4344. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4344`, October 2013.

[144] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[145] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *In 14th International Symposium on HighPerformance Computer Architecture (HPCA-14)*, 2008.

[146] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[147] Sharon Weinberger. Top ten most-destructive computer viruses. `http://www.smithsonianmag.com/science-nature/top-ten-most-destructive-computer-viruses-159542266/?all`, March 2012.

[148] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *IN NDSS*, 2003.

[149] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *Proceedings of the* $10^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '02, 2002.

[150] Ralph D. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[151] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 2014.

[152] Xilinx. Virtex-5 Power Spreadsheet, 2010.

[153] Xilinx. Xilinx ISE design suite 12.4, 2010.

[154] Xilinx. Xilinx UG190 Virtex-5 FPGA User Guide, march 2012.

[155] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings*

*of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, 2003.

[156] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, 2006.

[157] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, 2004.

[158] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. *Microarchitecture, IEEE/ACM International Symposium on*, 0:258, 2000.

[159] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, 2003.

[160] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, 2006.

[161] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI, 2008.

[162] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 367–377, New York, NY, USA, 2011.

[163] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent Value Locality and Value-centric Data Cache Design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, 2000.

[164] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, March 2009.

[165] Michael Zhivich and Tim Leek. Dynamic buffer overflow detection. In *In BUGS : Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[166] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, 2004.